RICE UNIVERSITY

Performance Optimizations for Software Transactional Memory

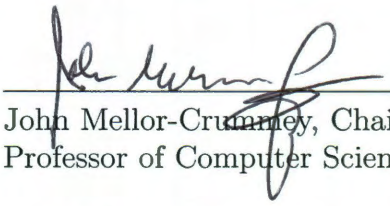
by

Rui Zhang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

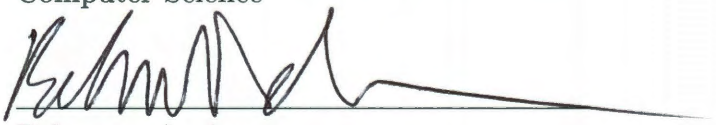


John Mellor-Crummey, Chair
Professor of Computer Science

William N. Scherer III, Co-advisor
Research Scientist of Computer Science

Zoran Budimlić, Co-advisor
Research Scientist of Computer Science

Keith Cooper
L. John and Ann H. Doerr Professor of
Computer Science



Behnaam Aazhang
J.S. Abercrombie Professor of Electrical
and Computer Engineering

Houston, Texas
August, 2010

ABSTRACT

Performance Optimizations for Software Transactional Memory

by

Rui Zhang

The transition from single-core processors to multi-core processors demands a change from sequential programming to concurrent programming for mainstream programmers. However, concurrent programming has long been widely recognized as being notoriously difficult. A major reason for its difficulty is that existing concurrent programming constructs provide low-level programming abstractions. Using these constructs forces programmers to consider many low level details. Locks, the dominant programming construct for mutual exclusion, suffer several well known problems, such as deadlock, priority inversion, and convoying, and are directly related to the difficulty of concurrent programming. The alternative to locks, i.e. non-blocking programming, not only is extremely error-prone, but also does not produce consistently good performance.

Better programming constructs are critical to reduce the complexity of concurrent programming, increase productivity, and expose the computing power in multi-core processors. Transactional memory has emerged recently as one promising programming construct for supporting atomic operations on shared data. By eliminating the need to consider a huge number of possible interactions among concurrent transactions, Transactional memory greatly reduces the complexity of concurrent programming and vastly improves programming productivity. Software transactional memory

systems implement a transactional memory abstraction in software. Unfortunately, existing designs of Software Transactional Memory systems incur significant performance overhead that could potentially prevent it from being widely used. Reducing STM's overhead will be critical for mainstream programmers to improve productivity while not suffering performance degradation.

My thesis is that the performance of STM can be significantly improved by intelligently designing validation and commit protocols, by designing the time base, and by incorporating application-specific knowledge. I present four novel techniques for improving performance of STM systems to support my thesis. First, I propose a time-based STM system based on a runtime tuning strategy that is able to deliver performance equal to or better than existing strategies. Second, I present several novel commit phase designs and evaluate their performance. Then I propose a new STM programming interface extension that enables transaction optimizations using fast shared memory reads while maintaining transaction composability. Next, I present a distributed time base design that outperforms existing time base designs for certain types of STM applications. Finally, I propose a novel programming construct to support multi-place isolation.

Experimental results show the techniques presented here can significantly improve the STM performance. We expect these techniques to help STM be accepted by more programmers.

Acknowledgments

It has been a long journey to arrive at this point. There are many people whom I am greatly indebted to during my years at Rice. Without them my thesis would never have come to true.

First and foremost, I owe my deepest gratitude to my late advisor Ken Kennedy. Ken is visionary, wise, humorous, friendly, and courageous. Without Ken's support and guidance, it would be impossible for me to pursue my research interests and complete these work. Ken will always be my role model and his invaluable heritage will always guide me in the future.

My deepest gratitude also goes to Zoran Budimlić and William N. Scherer III. I am extremely grateful to Zoran and Bill for their trust and giving me the freedom and guidance during my graduate studies. Zoran and Bill treated me as a peer and a friend with great patience and care. They not only taught me how to be a better researcher, but also how to be a better person.

I am also greatly indebted to John Mellor-Crummey for his invaluable inputs and his great help in the past year. John's deep and broad knowledge has been a great fortune for me. His engaging arguments and feedback have contributed greatly to this dissertation. John helped me better understand the limitations of my research and motivated me to find better ways to explain my results.

I would also like to thank my committee members, Keith Cooper and Behnaam Aazhang, for their insightful reviews and feedback that helped improve the quality of my work and my thesis.

I would also like to express my earnest gratitude to Vivek Sarkar. I was fortunate to have the privilege to work with Vivek on the work in chapter 8. I learned a great

deal from Vivek's deep and broad knowledge.

Thanks also go to Jan Hewitt for her generous help on the writing of my dissertation, which greatly improved the quality of my dissertation.

I am also greatly grateful to all my friends and colleagues with whom I spent my time as a graduate student at Rice and because of whom my time at Rice became fruitful and enjoyable. I would like to thank Mackale Joyner, Yuan Zhao, Yuri Dotsenko, Cristian Coarfa, Nathan Tallent, Cheryl McCosh, Todd Waterman, Ryan Zhang, Yi Guo, Jeff Sandoval, and Jason Eckhardt. My earnest gratitude also goes to all my other friends at Rice for their friendship these years.

I would also like to thank my grandparents, my parents, and my brother for their unconditional support. They have always been a source of energy for me. Without them, all my work would be meaningless.

Finally, I am deeply indebted to my dear wife, Jiayang, for her love, understanding, and support. Words cannot express how grateful I am for the time she has spent together with me. I dedicate this dissertation to her.

Contents

Abstract i

Acknowledgments iii

List of Illustrations ix

List of Tables xiii

1 Introduction 1

1.1 Single-core Processors vs. Multi-core Processors 2

1.2 Concurrent Programming 3

1.3 Transactional Memory 5

1.3.1 Performance Overhead of Software Transactional Memory . . 7

1.4 Thesis Overview 10

2 Background 14

2.1 Terminology 14

2.1.1 Compare-and-swap 14

2.1.2 Concurrency Control 15

2.1.3 Direct vs. Deferred Update 17

2.1.4 Strong vs. Weak Atomicity 18

2.1.5 Nested Transactions 19

2.2 Time-based STM 20

3 Related Work 23

3.1 Software Transactional Memory 23

3.1.1 Time-based STM Designs 26

3.1.2 STM Performance Optimizations by Relaxing Transaction Semantics . . . 30
3.1.3 Time Base Designs . . . 33
3.1.4 Other Related STM Research . . . 34
3.2 Hardware Transactional Memory . . . 34
3.2.1 Hybrid Transactional Memory . . . 36
3.3 Atomic Constructs . . . 36
3.3.1 Transactional Memory Implementation Approaches . . . 37

4 Runtime Tuning of STM Validation Techniques 39
4.1 Introduction . . . 39
4.2 Adaptive Validation Selection . . . 40
4.2.1 Threshold Hybrid Validation Strategy . . . 40
4.2.2 Implementation . . . 42
4.2.3 Benchmarks . . . 44
4.2.4 Runtime Tuning . . . 47
4.3 Evaluation of Runtime Validation Tuning . . . 48
4.3.1 Experiments based on RSTM 2 benchmarks . . . 49
4.3.2 Experiments based on STAMP benchmarks . . . 52
4.3.3 Discussion . . . 54

5 Commit Phase in Time-based STM 57
5.1 Existing Commit Phase Designs . . . 57
5.1.1 Transactional Locking II (TL II) . . . 57
5.1.2 Lazy Snapshot (LSS) . . . 59
5.1.3 Global Commit Counter (GCC) . . . 60
5.2 Commit Sequence Designs . . . 62
5.2.1 Non-unique Timestamps . . . 63
5.2.2 Unnecessary Update Avoidance . . . 65

5.2.3 Commit Sequence Design Alternatives 65
5.2.4 Theoretical Considerations 70
5.3 Experimental Results 77
5.3.1 Test Methodology 78
5.3.2 Discussion 79

6 Composability for Transactional Optimizations 84
6.1 Introduction 84
6.2 Composability Problem 85
6.3 Fast Read Interface Extension 93
6.3.1 Composability Mechanisms 94
6.3.2 Composability vs. Reuse 98
6.3.3 Composability Guarantees 99
6.4 Experimental Results 103
6.5 Discussion 106

7 Distributed STM Time Base 114
7.1 Introduction 114
7.2 Time Base Designs 118
7.3 Experimental Results 125
7.4 Discussion 126

8 Multi-place Isolation 133
8.1 Introduction 133
8.2 Single-place Atomicity in X10 136
8.2.1 Places 136
8.2.2 Atomic Blocks 137
8.2.3 Single-place Atomicity 138
8.3 Multi-place Isolation 139

8.4 Two-level Lock-based Place Isolation 142
8.5 Experimental Results 143

9 Conclusions **150**

Bibliography **154**

Illustrations

2.1 Compare-and-swap. 15

3.1 TL II validation. 28
3.2 Lazy snapshot validation. 28

4.1 Threshold hybrid validation. 41
4.2 VALIDATE(T). 41
4.3 RSTM metadata with addition for threshold-based validation. 43
4.4 Threshold hybrid strategy performance space. 45
4.5 Runtime tuning algorithm. 47
4.6 Runtime tuning result. The throughputs are normalized to the throughput of the runtime tuning strategy. 50
4.7 Throughput for six different benchmarks, at constant contention levels, with different number of threads. Where applicable, the x-axes show setups of the experiments. The setup includes the mix of operations and the number of threads. For example, 80/10/10 means 80% lookups, 10% inserts, and 10% removes. 2T means using 2 threads. The throughputs shown are transactions per second and are normalized to the throughput using our runtime tuning strategy. . . 51

4.8 Execution time and speedup for Labyrinth and Intruder. The bars represent execution times. The lines represent performance speedup of results using our runtime tuning strategy over TL II and LSS. The y-axis on the left is execution time. The y-axis on the right is speedup percentage. 53

5.1 TL II COMMIT(T) single counter version. 58
5.2 TL II COMMIT(T) tuple version. 59
5.3 TL II TX_READ(T, O_i, m) single counter version. 59
5.4 TL II TX_READ(T, O_i, m) tuple version. 60
5.5 Lazy snapshot COMMIT(T). 60
5.6 Lazy snapshot TX_READ(T, O_i, m). 61
5.7 GCC TX_READ(T, O_i, m). 61
5.8 GCC COMMIT(T). 62
5.9 GCC TRY_COMMIT(T). 62
5.10 V1, V2, V4 TX_READ(T, O_i, m). 67
5.11 V1 COMMIT(T). 68
5.12 V2 COMMIT(T). 69
5.13 V3 TX_READ(T, O_i, m). 69
5.14 V3 COMMIT(T). 70
5.15 V4 COMMIT(T). 71
5.16 LinkedList, 80% lookup/10% insert/10% remove. 79
5.17 HashTable, 33% lookup/33% insert/33% remove. 80
5.18 LinkedListRelease, 80% lookup/10% insert/10% remove. 81
5.19 RBTree, 80% lookup/10% insert/10% remove. 82
5.20 RandomGraph, 80% lookup/10% insert/10% remove. 83

6.1 Labyrinth transaction pseudocode. 86

6.2 Insert - pure TM version. 88
6.3 Insert - uncomposable version. 89
6.4 A composed list insertion example. 90
6.5 A composed conditional list insertion example. 90
6.6 A hidden update example. 91
6.7 Fast read in lookup scheme. 95
6.8 Flush in the lookup scheme. 95
6.9 Fast read in PCM scheme. 96
6.10 Flush in PCM scheme. 97
6.11 Labyrinth execution time results. The figure shows the execution times of Labyrinth with different thread numbers using three different schemes. 107
6.12 Genome execution time results. The figure shows the execution times of Labyrinth with different thread numbers using three different schemes. 107
6.13 Vote execution time results. The figure shows the execution times of one million transactions of Vote with different thread numbers using three different schemes. 108
6.14 Set execution time results. The figure shows the execution times of one million transactions of Set with different thread numbers using three different schemes. 109
6.15 Nested set execution time results. The figure shows the execution times of one million transactions of nested set with different thread numbers using three different schemes. 110
6.16 Intruder execution time results. The figure shows the execution times of Intruder with different thread numbers using three different schemes. 111

7.1 Throughput comparison of the Time_Base_Test benchmark using a distributed time base and a centralized time base. The figure shows that the distributed time base outperforms the centralized time base. 115
7.2 Time_Base_Test implementation. 115
7.3 Definition of a cache line size int. 120
7.4 Distributed time base metadata. 122
7.5 The open function when using the distributed time base design. . . . 126
7.6 Bayes performance results. 127
7.7 Genome performance results. 128
7.8 Intruder performance results. 128
7.9 Kmeans performance results. 129
7.10 Labyrinth performance results. 130
7.11 SSCA2 performance results. 131
7.12 Vacation performance results. 132

8.1 Naive linked list Insert with multi-place isolated statement. 139
8.2 Optimized linked list Insert with multi-place isolated statement. . . . 141
8.3 Implementation of isolated(*). 142
8.4 Implementation of isolated(⟨place list⟩). 143
8.5 Multi-place based sorted LinkedList implementation. 144
8.6 Throughput on Gamma - a 64-way Niagara 2 system. 147
8.7 Throughput on Swym - a 16-way SunFire 6800 system. 148
8.8 Throughput on Sugar - an 8-way Intel Xeon system. 149

Tables

5.1 Commit sequence comparison. 66

Chapter 1

Introduction

This thesis proposes four novel techniques to improve the performance of software transactional memory, a promising construct for concurrent programming, which currently suffers from high performance overhead; and a new programming construct to support multi-place isolation.

Concurrent programming has existed for decades and is well known to be difficult. A correct and efficient concurrent program requires deep understanding of concurrency and careful attention to a huge number of interactions among concurrent activities. Concurrent programming is error-prone, and concurrency bugs are extremely difficult to identify because they depend on the interleaving of thread actions, which are not readily repeatable. Until recently, most programmers only needed to write sequential programs and were free of the problems haunting the writing of concurrent programs. Unfortunately, the hardware industry has evolved to the state where it has to resort to multi-core processors for further performance advancement. The ubiquitous existence of multi-core processors requires not only the elite few who mastered concurrent programming but also ordinary programmers to face the challenge. This change greatly shakes the foundation of existing programming practice and demands a thorough review of existing programming tools.

Mutual exclusion has been a critical operation in concurrent programming. The current dominant programming construct for achieving mutual exclusion, locks, have several well known problems that are directly related to the difficulty of concurrent

programming. Transactional memory (TM) has emerged as a promising technique for mutual exclusion in recent years. But existing transactional memory systems suffer from high performance overhead that prevents it from being widely used. The techniques in my thesis are designed to reduce the performance overhead of software transactional memory (STM) [Fra03, GHP05, HF03, HPST06, HLMS03, MSH+06, SATH+06] and make it more usable for ordinary programmers.

My thesis is that the performance of STM can be significantly improved by intelligently designing validation and commit protocols, by designing the time base, and by incorporating application-specific knowledge. Improving the performance of software transactional memory is important for broadening its acceptance among software developers.

1.1 Single-core Processors vs. Multi-core Processors

Unlike multi-core processors, superscalar single core processors provide a sequential programming abstraction. This abstraction hides the underlying complexity of instruction level parallelism (ILP) [OH05] from programmers. The sequential illusion enables programmers to write code without understanding parallelism on the instruction level. In the past three decades, the improvement of architecture and fabrication technology was able to keep improving single-core processor performance 40% to 50% annually on average [OH05]. This gave programmers the convenience of enjoying performance improvement by simply using a more powerful processor for their sequential applications without rewriting their code.

Unfortunately, the semiconductor industry has evolved to such a point that improving single core processor performance at a similar pace as before has become infeasible. The amount of heat generated by higher frequency processors and the

diminishing return from architecture innovations [OH05] slowed the advancement of single-core processor performance. Without devoting vast resources into dissipating the generated heat, it becomes very difficult to increase the processor's frequency. As a result, processor's frequency can only be improved at a much slower pace than before. Also, architectural innovations now only bring less and less performance returns [OH05]. But on the other hand, The number of transistors on a single die is still expected to double approximately every two years – at a pace described by *Moore's law* [OH05]. Based on the above facts, the industry has resorted to *multi-core* processors, which pack multiple cores in one processor. Without increasing the frequency of the processor, more computing power is packed within one processor. Naturally, more cores can be put inside a single processor when the number of available transistors increases and the peak computing power of one processor linearly increases with the number of cores.

There is one critical difference between the parallelism on the core level and the parallelism on the instruction level in superscalar processors. The difference is that there is no sequential illusion for the core level parallelism. To efficiently use these multi-core processors, programmers need to manage the core level parallelism explicitly and this raises the big question of how to program these multi-core processors.

1.2 Concurrent Programming

The lack of sequential abstraction on the core level of multi-core processors imposes an immediate challenge on programmers. The way of improving the performance of an application is no longer tuning its sequential algorithm or running it on a more powerful computer. Instead, the programmers need to explicitly split their applications into concurrent tasks to effectively utilize multiple cores.

Synchronization is necessary to coordinate concurrent operations on shared data to avoid corruption, because uncoordinated concurrent operations can lead to nondeterministic results and break data consistency. For example, the final value of a shared variable after two uncoordinated concurrent updates can be from either update or some combination of the two. Current concurrent programming is based on a few low level programming abstractions such as locks, semaphores, and monitors. For example, locks are used to provide mutual exclusion so that one can update shared data without others seeing an inconsistent view of the data. But programming based on these constructs makes programmers think at a very low level of concurrent programming. For example, when using locks, a programmer needs to worry about which piece of memory is protected by which lock; the granularity of locks; and when and where to acquire and release locks to avoid deadlock or live lock. The necessity of considering a great number of possible interactions among concurrent activities also adds a lot of complexity. Moreover, a programmer needs to imperatively encode the way of using these constructs in their applications, though this part of the code might not be an integral part of the application's semantics. For example, if a programmer wants to write a concurrency-safe queue using locks, the programmer needs to carefully design the locks that protect the queue. He needs to choose whether to use a big lock to protect the entire queue or use small locks to protect each node of the queue. Then the programmer needs to write the code that uses these locks to ensure correctness. But the semantics of the queue are unrelated to these lock operations. The part of code operating the locks is purely a byproduct of using locks and is forced to be written when programming is based on locks. Furthermore, the code becomes difficult to read and maintain due to the tightly coupled application semantics and concurrency control. Even worse, lock-based code is not composable

when the programmer wants to reuse it to write new applications. A programmer has to understand the internal implementation details of lock-based code before he can correctly reuse it in new applications.

1.3 Transactional Memory

The aforementioned difficulties of concurrent programming and the necessity of writing concurrent programs for multi-core processors underscores the need for better constructs for concurrent programming. Extensive research has been conducted to design new programming abstractions that can reduce the concurrent programming complexity. Among these research efforts, Transactional Memory (TM) [HM93] has emerged as one promising programming construct for mutual exclusion. A memory transaction is a sequence of memory operations that either executes completely or has no effect. Code enclosed within a transaction appears to execute instantaneously and indivisibly to an outside observer. Using TM, a programmer only needs to enclose a piece of code that needs to be concurrency safe inside a transaction and the TM system will guarantee the correctness. Programming using TM is a much simpler effort than using locks because the programmer does not need to worry about how a transaction interacts with other concurrent transactions and only needs to know which part should be in a transaction. Not needing to consider the vast number of interactions greatly reduces the complexity of concurrent programming.

Transactions are not a new concept. They have been widely used in the database community. A *transaction* is an abstraction of a sequence of operations that appears to happen indivisibly and instantaneously to an outside observer. The abstraction of transactions hides the internal operational details of a transaction from an outside observer and makes reasoning about transactional correctness much easier than with

lower level primitives such as locks. Similar to database transactions, transactions in transactional memory support the properties of *atomicity, consistency, and isolation.* But unlike database transactions, transactions in TM do not support *durability.*

Atomicity is a property that requires a transaction to either successfully complete all its operations or to appear not to have executed. A transaction that successfully completes all its operations is called committed; a transaction that appears not to have executed is called aborted.

Consistency requires a transaction to keep the state of relevant shared data structures consistent after it commits. The definition of being consistent for a data structure depends on the semantics of the data structure and is usually defined by a set of invariants. To support consistency, transactions need to always start with consistent states and also leave with consistent states when transactions commit. The consistency property does not require the internal states of a transaction be consistent.

Isolation is a property that requires a transaction's correctness not to be compromised whether it is running alone or running concurrently with other transactions.

Durability requires a transaction's changes to last across applications, which means that once a transaction commits, the changes it made stays available for future applications. The argument that memory transactions do not support durability is based on the assumption that changes made by a memory transaction are only alive within its owner application.

Memory transactions are also composable. This means a programmer can compose smaller transactions into bigger transactions without knowing the details of the transactions. In comparison, programs based on locks are not composable. Simply putting lock-based procedures together does not guarantee correctness. For lock-based code, the programmer needs to understand the implementation details for reusing them.

Despite the nice properties, adding TM as a programming construct faces many challenges. For example, being a programming construct, transactional memory has to retrospectively co-exist with existing programming features. Integrating transactional memory and these features is one problem that must be addressed. Moreover, transactional memory incurs some performance overhead compared with fine-tuned lock-based implementation or non-blocking algorithms. Though it is not surprising to incur extra overhead with extra property guarantees, it is crucial to reduce the overhead to an acceptable level for TM to be widely used.

1.3.1 Performance Overhead of Software Transactional Memory

Though transactional memory has the capability of greatly simplifying concurrent programming, its performance overhead is still high. Some applications based on software transactional memory, transactional memory systems implemented purely in software, still exhibit high performance overhead compared with their sequential, fine grain locking based, or non-blocking based implementations. The performance overhead of STM systems comes from several aspects. For example, as noted in [ZBS10], STM systems must perform additional work to guarantee transactions' consistency, while this can be trivially achieved in sequential programs. To provide this guarantee, STM systems need to record sufficient information, such as transactional reads, to validate the transactions when needed. This additional work can sometimes entirely negate the initial purpose of using a parallel implementation: better performance [YNW+08]. The argument for parallelization does not hold if the performance of a parallel version trails that of a sequential counterpart that takes less effort to develop. Though a lot of research has been conducted on designing efficient STM systems to reduce the overhead, the performance of software transactional

memory is still far from being solved. In chapter 4, I present a new runtime tuning technique that focuses on improving the performance of the validation technique of STM systems.

Although part of the overhead of STM, such as consistency checking in general, is unavoidable due to the nature of concurrency, some overhead occurs because of conservative assumptions made by the STM systems. Designed as a general mechanism, STM systems lack any application-specific knowledge. This lack forces them to perform more work that a more application-specific approach could avoid. For example, many STM systems keep a set of past transactional reads [HLMS03, MSH+06] and validate that a transaction is still consistent by checking whether any of these variables has changed. If any of the past reads has changed, the transaction is deemed inconsistent and is aborted. This type of validation is conservative and gives false positives in many execution scenarios, since a change of a value that has been previously read by a transaction does not necessarily imply an order cycle among transactions. For example, in this type of STM systems, if a transaction A reads variable x then updates variable y and another transaction B modifies variable x after A reads and before A commits, A will abort when it detects the fact that x has been modified. But actually both A and B could commit. Therefore these STM systems abort many transactions that could have been allowed to complete. In chapter 6, I present a novel STM extension that enables an STM application to take advantage of application-specific knowledge while maintaining transactions' composability.

Timestamp-based STM systems such as transactional locking II [DSS06] often assume that if an object being opened is newer than the transaction's timestamp, a conflict has occurred, another conservative design. Some systems [RFF06] mix the use of timestamp- and list-based validation to reduce the number of false positives, but

still cannot completely eliminate false conflicts. Also, existing timestamp-based STM systems use unique timestamps for each update transaction and operate the time base in a way that usually makes false updates. Both the unique timestamp and false updates make conservative assumptions about the execution and may deteriorate the performance. In chapter 5, I propose four new commit sequences for timestamp-based STM systems to solve both or one of these problems. Also, existing timestamp-based STM systems use a centralized integer counter as the time base which is a potential scalability bottleneck. We propose a distributed time base design to solve this scalability problem in chapter 7.

Another part of the overhead comes from the optimistic nature of STM systems. The STM system allows multiple transactions to run concurrently, with the assumption that they may not conflict. But transactions do conflict and thus abort. Also, the amount of conflicts often increases with the number of threads concurrently running in the system, which leads in turn to yet more wasted work.

An outcome of these performance related issues is that, for some applications, the overhead of the STM system can be high enough to make programming based on STM less desirable. Consequently, for certain applications, the programmer may have enough performance gain incentive to design custom consistency guarantee mechanisms. This would effectively reduce part of the overhead associated with the STM system. Importantly, though the performance gains can be substantial, the work to apply such optimizations is not trivial and requires good understanding of both the application and of concurrent programming.

1.4 Thesis Overview

Chapters 4 through 7 of my thesis focus on problems on the performance aspect of software transactional memory (STM). I show that STM system's performance can be effectively improved by certain optimizations and understanding application characteristics, and that there are many performance tradeoffs that should be carefully evaluated when designing a STM system. There are several reasons that make improving performance of STM important. First, the overhead of existing STMs is still high and it is necessary to reduce the overhead. Sometimes the overhead is high enough to entirely negate the performance incentive of writing a concurrent counterpart of a sequential application. Second, the performance of STM applications is sensitive to different TM implementations. There are many tradeoffs that the underlying designs need to consider and most times there is no globally best technique. The dependence of performance over these tradeoffs requires understanding various STM design choices. Third, understanding STM can help with the design of critical hardware support to improve TM performance. In chapter 8, I propose a novel multi-place atomicity programming construct to address the scalability challenge of enforcing atomicity when operating on distributed data in multi-core processors. Our solution extends the X10 *place* construct with *multi-place atomic statements.* We introduce a *two-level lock-based* implementation of the multi-place atomic construct. The resulting language construct is as easy to use as transactions or coarse-grain locking, while it yields performance that is on par with fine-grain locking.

Next is a brief overview of my thesis.

- Incremental validation has been one major source of performance overhead in early STM systems such as DSTM, RSTM [HM93]. Among various techniques

designed to reduce the overhead, time-based STM systems show effective improvement of the performance. Timestamp-based software transactional memory validation techniques use a global shared counter and timestamping of objects being written to reason about sequencing of transactions and their linearization points, while reducing the number of unnecessary validations that have to be performed, thus improving overall system performance. Transactional locking II [DSS06] and lazy snapshot [RFF06] validation techniques have so far shown the best performance for a wide variety of applications. Unfortunately, the performance of these two techniques depends heavily on the application: the number of concurrent jobs, the length of the transaction and the read/write ratio of the shared objects can significantly favor one technique over the other. Moreover, for long-running applications that change their behavior over time, neither of these two techniques is optimal. I propose a runtime tuning strategy that uses profiling to determine the most profitable validation technique. Our runtime tuning strategy can behave as an arbitrary mix of transactional locking II and lazy snapshot techniques depending on the state of the STM system. This work will be further discussed in chapter 4.

- As noted in [ZBS08a], during the commit phase of a timestamp-based validation scheme, several actions have to be performed: locking of the objects being written to the memory, atomically incrementing a shared timestamp counter, updating timestamps for objects being committed to memory, performing a final validation of the transaction's consistency, and atomically effecting the transaction's changes to the outside world. The order and manner in which these actions are performed can affect both the correctness of the STM implementation and the overall system performance. We identify several commit

sequence designs, prove their correctness, and analyze their performance. We identify cases where timestamps do not have to be unique for different transactions committing concurrently, and cases where unnecessary updates of the global shared counter — which can trigger extra validations in other transactions, hurting performance — can be avoided. This part of work will be further discussed in chapter 5.

- To reduce the performance overhead of STM, programmers sometimes carefully bypass certain TM calls based on application-specific knowledge, as noted in [ZBS10]. Such optimizations usually break *isolation* [LR06] and *consistency* [LR06], two key properties of transactions, so that the programmer has to encode by hand consistency checking into the application. However, this type of optimizations can also break *composability* [LR06], another desirable property of transactions; nesting transactions optimized in such a way can lead to incorrect results. In chapter 6, we identify the composability problem associated with optimizing software transactional memory application performance using application-specific knowledge. We also propose an extension to STM systems that solves this problem. This proposed extension offers comparable flexibility in optimizing STM performance while maintaining the composability.

- Time-based STM systems use timestamp information to reduce the number of expensive operations on validating transactions' past reads at runtime. A critical component of time-based STM systems is the time base that is responsible for generating timestamps for all transactions in the system. The way a time base functions is closely related to the performance of the STM system. Existing time base designs in STM systems use a single integer as the time base

shared by all transactions. Every timestamp operation needs to consult the time base for the current time or acquiring a new timestamp. The centralized nature of this type of time base is potentially a scalability bottleneck. In chapter 7, we present a distributed time base design as an alternative for addressing this scalability bottleneck.

- The `atomic` statement in X10 guarantees *single-place atomicity* because a task (activity) is only permitted to access place-local data within the `atomic` statement. Though it provides a sound semantics, it is restrictive for computations where an atomic statement may need to access data located in more than one place. In chapter 8, we extend the X10 model with *multi-place isolated statements* of the form `isolated(`⟨*place-list*⟩`)` and `isolated(*)`, and generalize X10's locality rule [CGS+05] by requiring that all data accessed within a multi-place isolated statement be local to any one of the places in the statement's scope.

Chapter 2 presents some background information of my thesis. Chapter 3 reviews the related work of my thesis. In chapter 4, I discuss an adaptation technique for time-based validation mechanisms. Chapter 5 shows several commit phase designs of time-based STM systems. Chapter 6 describes an STM interface extension to add composability support for STM optimizations based on raw shared memory reads. Chapter 7 proposes the distributed time base design for time-based STM systems. Chapter 8 presents a design for a multi-place isolation construct for X10. Chapter 9 presents conclusions from this research.

Chapter 2

Background

In this chapter I first introduce some terms used in this dissertation, and then I will give a background introduction of time-based STM.

2.1 Terminology

2.1.1 Compare-and-swap

Compare-and-swap (CAS) is a hardware primitive widely used in building concurrent data structures. Figure 2.1 shows pseudocode for CAS. CAS takes three arguments: a memory location A, a value B that is expected to be read from the location A, and a new value C to write to location A if B is found there. If location A contains a value different from B, CAS returns the value it found in location A without updating A. CAS's verification of location A is value based, which means that it can not detect the "ABA" pattern, in which the value in a location changes to another one and then changes back to the original one. Most modern processors support CAS in hardware. A typical use of CAS is to read a memory location, perform some computation on the value, and then try to put the new value back to the location using CAS. If CAS succeeds it ensures that the location's value has not been altered in the meantime. An alternative to CAS, load-link/store-conditional [Her90], can be used to perform the same function as CAS.

```
atomic word CAS(word *ptr, word old, word new)
{
    word result = *ptr;
    if (result == old)
        *ptr = new;

    return result;
}
```

Figure 2.1 : Compare-and-swap.

2.1.2 Concurrency Control

A concurrency control mechanism is one that coordinates concurrent accesses to shared resources. When multiple activities compete to access the same resources concurrently without coordination, the end result depends on the order of these activities and is nondeterministic. For most applications, it is necessary to manage concurrent activities in a way that makes reasoning about the result possible. A concurrency control mechanism is designed to coordinate conflicting accesses. It is worth mentioning that the nondeterministic nature of conflicting accesses may be benign for some applications depending on their semantics. For this type of application, concurrency control is not necessary.

For a conflict to occur there must be more than one accesses to the same memory location at the same time and at least one of these accesses must be a write. Concurrent transactions must detect and resolve conflicts to provide sound transactional semantics. In total, there are three types of possible conflicts: read-after-write, write-after-write, and write-after-read. For a TM system, all of these conflicts between transactions must be detected and resolved. Conflict detect mechanisms can be cate-

gorized into two types: *eager conflict detection* and *lazy conflict detection.* One thing worth pointing out here is that conflicts between transactions do not necessarily lead to transaction failure; all relevant transactions involved in a conflict may commit successfully if their conflicts does not violate the correctness criteria of transactions. One correctness criterion widely used in TM is *serializability* which requires the result of a concurrent execution to be the same as the result of a serial execution that interleaves the same transactions. For eager conflict detection, conflicts are detected at the moment when they occur. For example, *visible reads* in STM detect write-after-read when writes happen and allow eager write-after-read conflict detection. Also, acquiring the lock for a memory location on its first access allows eager write-after-write and read-after-write conflict detection by enabling other transactions verify whether the location is locked on their first accesses.

For lazy conflict detection, conflicts are detected at a time later than the point when the conflict actually occurs. For example, *invisible reads* do not detect write-after-read conflicts when writes happen and postpone the detection of write-after-read conflicts to when a transaction verifies its past reads. Another example is that *versioning* can postpone the detection of read-after-write and write-after-write conflicts. To enable lazy conflict detection, necessary information needs to be kept for later verification. Any time during a transaction can be chosen to do conflict detection; existing TM systems typically performs it when a new object is opened or in the commit phase.

Conflicts need to be resolved after they are detected. Resolving a conflict in TM is an arbitration process, determining if a conflict violates transactional correctness criteria. When resolving a conflict, a TM system may abort or delay the execution of one of the conflicting transactions. Conducting both conflict detection and resolution

at the same time when the conflict occurs is often referred as *pessimistic concurrency control*. Delaying the detection and resolution to a later time is referred to as *optimistic concurrency control*. Pessimistic concurrency control resolves conflicts early and can reduce wasted work from transactions that are aborted later. But it may abort some transactions that could commit eventually. Comparatively, optimistic concurrency control permits conflicting transactions to proceed. It may increase the amount of wasted work if a transaction is eventually aborted, but gives transactions a better opportunity to commit.

Based on the way a TM handles read-write conflicts and write-write conflicts, the concurrency control mechanisms in TM may be categorized as the follows [MSS04].

- Invisible vs. visible reads: For invisible reads, a TM system does not detect write-after-read conflicts when a transaction declares its intent to update an object that another concurrent transaction has read. For visible reads, detection of write-after-read conflicts happens when the transaction declares its intent to update an object.

- Eager vs. lazy acquire: An eager acquire TM system detects write-after-write and read-after-write conflicts on open, when a transaction declares its intent to access an object. In contrast, lazy acquire detects these conflicts at commit time.

2.1.3 Direct vs. Deferred Update

The difference between direct and deferred update is whether changes are written directly to an object or to some secondary location before they are committed. In a TM system that uses *direct update*, a transaction updates the object directly, but for

a TM system that uses *deferred update*, a transaction keeps the update to an object somewhere else and writes these changes back to the object later. Direct update usually has a faster commit phase compared to deferred update, because deferred update needs to write the updates to the object or replace it with a newer copy. On the other hand, using deferred update performs aborts faster, because it can simply discard changes while direct update needs to revert the object.

2.1.4 Strong vs. Weak Atomicity

Strong atomicity and weak atomicity [BLM05] define how the memory references outside a transactional region interact with the code inside a transactional region. Strong atomicity guarantees transactional semantics between transactions and non-transactional code. Weak atomicity only guarantees atomicity among transactions.

In strong atomicity semantics, any memory reference outside a transaction follows the protocol of operations inside a transaction region. The consistency of a transaction is not affected by any outside memory operations. In weak atomicity semantics, memory references outside a transaction do not follow the transactional protocol. As a consequence, the consistent memory view for a transaction may be interrupted by an outside conflicting memory reference. The responsibility for isolating the transactional region and non-transactional regions falls to the programmer.

Strong/weak atomicity vs. *place-based atomicity.* There is an important difference between the strong/weak atomicity and place-based atomicity: whether the entire, global memory space is visible to a transaction, or only some partitions. Both strong and weak atomicity implicitly use a global view of the shared memory space. For strong atomicity, any conflicting memory reference outside a transaction is guaranteed to not affect the correctness of a transaction. For weak atomicity, any conflicting

memory references outside a transaction could affect the correctness of a transaction. For strong atomicity, the TM system is entirely responsible for managing the conflicts between transactions and outside shared accesses; for weak atomicity, both the TM system and the programmer take the responsibility of managing the conflicts. In contrast, with place-based atomicity semantics, atomicity at the global level is provided with an extra guarantee – the place system. By adding the place system as a new component for ensuring global level atomic semantics, we enable additional optimizations in TM systems and compilers.

2.1.5 Nested Transactions

A nested transaction is a transaction whose execution takes place entirely within the context of another transaction. Internal state of the enclosing transaction is visible to its nested transactions. Nested transactions can be categorized into three types based on the semantics of the nested transactions.

- *Flattened nested transaction*: A flattened transaction is essentially an inlined transaction. The inner transaction is inlined into the outer transaction and performs as a part of the outer transaction.

- *Closed nested transaction*: If a closed nested transaction aborts, it does not abort its enclosing transaction. When a closed transaction commits, its changes are only initially visible to its surrounding transactions; they are only made globally visible when the outer-most surrounding transaction commits.

- *Open nested transaction*: If an open transaction commits, its changes are made globally visible to non-enclosing transactions immediately. Open nested transactions requires extra programming effort for correctness guarantees.

2.2 Time-based STM

This section introduces *time-based* STM systems. Time-based STM systems use time information embedded within transactions and shared objects to detect conflicts and to guarantee consistency. Usually each shared object is associated with a timestamp that indicates the time the object was last modified. Every time an object is updated, its timestamp is updated. A timestamp is also associated with each transaction. Comparing a transaction's timestamp and a shared object's timestamp enables a time-based TM system to detect potential conflicts. It is worth noting that consistency checking in a time-based TM system also generates false positives that abort transactions that could have committed. Compared with validation, time-based STM systems are more efficient due to low overhead, though time-based STM systems incur some space overhead for keeping timestamps.

- Lev and Moir[LM04] propose a heuristic, *conflict counter*, to reduce validation overhead. Their heuristic uses per object read counters and a global conflict counter. A transaction increments the read counter of an object when opening it and decrements the read counter when committing the transaction. The STM system also maintains a global conflict counter which is incremented whenever a transaction needs to open an object with non-zero read counter for writing. A transaction keeps a snapshot of the conflict counter and can skip validation when opening a new object if the conflict counter is the same as the snapshot value. The conflict counter heuristic is able to reduce the number of object validations. The disadvantage of this heuristic is every read operation now needs to perform an atomic add operation on a read counter and this can lead to heavy cache coherence traffic. Also the heuristic is very pessimistic. A transaction performs

validation for all conflict counter changes it detects but many of these changes are from totally irrelevant transactions.

- In Spear et al.'s [SMSS06] *global commit counter* (GCC) heuristic, a transaction does not perform a validation if no writes were committed in the entire system since the last object was opened. This scheme can avoid many unnecessary validations, leading to performance improvement for situations where writes happen rarely in the system. However, this strategy is very conservative: a write to a shared object does not affect a transaction's consistency if that object is never used in that transaction, yet GCC will treat it as a conflict and validate the shared objects the transaction has read.

- Dice et al.'s [DSS06] *transactional locking II* (TL II) algorithm is another time-based STM implementation. Their algorithm associates a timestamp with every shared object at the time of its modification. A transaction acquires its own timestamp once at the beginning of execution, and commits successfully if none of the objects opened during the execution has a timestamp newer than the timestamp of the transaction. A transaction aborts if it encounters an object with a newer timestamp. An advantage of this algorithm is that it can completely eliminate the bookkeeping overhead of read-only transactions since it never validates opened objects, making it attractive in situations where contention is low and most transactions commit successfully.

- *Lazy snapshot* (LSS) [RFF06] algorithm is another time-based STM implementation. This algorithm associates a timestamp with every shared object at the time of its modification. A transaction acquires its own timestamp once at the beginning of execution, and commits successfully if none of the objects

opened during the execution has a timestamp newer than the timestamp of the transaction. Unlike TL II, LSS does not abort a transaction immediately if it encounters an object with a newer timestamp. Instead, LSS performs a validation over its past reads by verifying if none of its past reads is modified by other transactions. The transaction continues if the validation succeeds and is aborted otherwise. This approach requires the transaction to keep its past reads, but it has a higher probability of committing a transaction.

Chapter 3

Related Work

In this chapter, I summarize previous work related to my thesis. First, I present the related work in software transactional memory (STM). Then, I discuss the related work in hardware transactional memory (HTM). Last, I cover work related to the multi-place atomicity.

3.1 Software Transactional Memory

Software transactional memory is transactional memory implemented purely in software. Comparatively, hardware transactional memory requires new hardware support. No need of new hardware support makes STM more flexible than HTM. For example, STM can be used on existing hardware architectures, but HTM cannot. On the other hand, STM's performance is usually worse than HTM.

The first STM system is proposed by Shavit and Touitou [ST95]. Only static transactions are supported in their system, which means a transaction needs to declare all memory locations it might access in advance. Their system uses two-phase locking to acquire ownership of shared objects. A transaction commits after acquiring ownership over all of its memory locations. To avoid deadlock, their STM system acquires ownership in a predetermined order. Their system offers a lock-free progress guarantee which means that some transaction will succeed after a finite number of steps, even if there exist thread failures. The lock-free guarantee is implemented using

helping that allows a transaction A to help execute another transaction B when A can not obtain the ownerships from objects owned by B.

The requirement of declaring memory locations in advance limits the usage of Shavit and Touitou's TM system. Later, Herlihy et al. developed the first dynamic STM - DSTM [HLMS03]. Unlike Shavit and Touitou's system, DSTM does not require a transaction to declare the memory locations it might access in advance. DSTM uses obstruction freedom as its progress guarantee. Obstruction freedom is weaker than lock freedom or wait freedom. This reduces the implementation complexity and provides higher performance. DSTM also introduced an explicit contention manager to coordinate the conflicts among transactions. DSTM is an object level granularity STM system. DSTM uses optimistic concurrency control and uses non-blocking synchronization. DSTM is a deferred-update system. DSTM also proposes a technique to improve STM performance via "early release", which allows a transaction to release an object before committing. Early release can reduce the validation overhead and increases the chance to commit at a cost of increased programming complexity. In DSTM, a transaction references an object through a `TMObject`. Every access to a transactional object is through its `TMObject`. Within each TMObject, there are the pointers to its transaction descriptor, the old version of the object, and the new version of the object. DSTM creates a shadow copy of an object to modify and commits the shadow copy when successful.

Many dynamic STM systems have been developed afterwards. Harris and Fraser designed the WSTM system [HF03]. WSTM is a dynamic word granularity system and provides obstruction free progression guarantee. Fraser designed OSTM [Fra03] that uses deferred update and provides a lock free progress guarantee in his thesis.

Performance has been an important topic of research. Early STM systems pro-

vide rather poor performance compared to fine grain locking based implementations or non-blocking based implementations. Marathe et al.'s ASTM [MSS05] optimizes DSTM by eliminating one redundant pointer indirection for read only objects. They also designed an adaptive strategy that allows a transaction to choose between early conflict detection and late conflict detection. Marathe et al. designed RSTM [MSH+06]. It contains several performance enhancements for a deferred-update STM. RSTM removes one level of pointer indirection compared to DSTM and ASTM. RSTM also provides its own memory allocator so RSTM can work with a language without garbage collection like C++.

Later research showed that lock-based implementations have the potential to outperform non-blocking STM systems due to less wasted work and easier implementations. Dice and Shavit designed an STM system called transactional locking (TL) [DS06]. TL locks the objects at commit time and so have a smaller conflict window compared with locking on first access. TL uses a deferred update scheme, so it does not lock the shared objects at first access like other lock-based STM systems. McRT-STM [SATH+06] developed by Saha, et al. uses two-phase locking rather than a non-blocking scheme. Their system uses timeouts to detect deadlocks.

People also investigated using direct update instead of cached update to improve the commit path's performance. In direct update, a transaction can commit successfully without writing cached copies to the memory. Direct updates create a faster commit phase but incur more overhead when a transaction aborts. McRT-STM is a direct update system. BSTM [HPST06] developed by Harris et al. is also a direct-update STM. BSTM uses two-phase locking to get exclusive accesses to objects being updated. It uses versioning to guarantee consistency. McCloskey et al.'s Autolocker [MZGB06] uses pessimistic concurrency control and uses two-phase locking

to lock every shared read and write when committing a transaction. Their system prevents deadlock by acquiring the locks in a well-defined order.

How to manage contention is one problem some research focused on, especially for STM system with obstruction-free progress guarantee. Contention management is necessary to coordinate conflicting shared memory accesses. Scherer and Scott investigated various contention management policies [SS05]. Contention managers determine which transaction may proceed when a conflict occurs and which transaction should abort. Guerraoui et al. designed SXM [GHP05] that allows a transaction to dynamically choose its contention management policy.

STM research also considers optimizing other STM overheads. Spear, Michael and Praun designed RingSTM [SMv08]. RingSTM focuses on reducing the overhead associated with operating transactional meta data. It compresses read and write sets using a Bloom filter and uses a centralized coordination mechanism for commits that commits a transaction by enqueuing its filtered write set onto a global list. RingSTM is able to commit a transaction with only a single atomic compare-and-swap operation. Comparatively, other STM need to use more compare-and-swap operations to acquire ownerships of shared object accessed by a transaction. The disadvantage of RingSTM is the increased probability of aborting a transaction because the conflict detection is entirely determined by the Bloom filter that can not precisely represent the information of what memory locations have been accessed.

3.1.1 Time-based STM Designs

One area closely related to STM system performance is the technique for guaranteeing the transaction consistency. *Incremental validation* [HLMS03] is a strategy widely used in early STM systems such as DSTM and RSTM. *Incremental validation* verifies

all past invisible reads and lazy writes every time the transaction opens a new object. If any change in the past is detected, the validation fails. This strategy guarantees a consistent state but imposes a substantial overhead [SMSS06], since it is essentially a $O(n^2)$ operation where n is the number of objects opened in a transaction. *Incremental validation* is widely used in early STM systems that use optimistic concurrency control and delayed read-write conflicts detection. It ensures that transactions always operate on consistent states and avoid exceptions such as infinite loops or null pointer dereferences. Experimental results have shown that incremental validation incurs high performance overhead and becomes a major source of STM performance loss [CBM+08].

Time-based validation is a strategy developed to reduce the overhead of guaranteeing consistency using incremental validation. Time-based validation strategies guarantee the consistency of the past reads by simply checking whether the timestamp of the object being opened is in the transaction's validity range. This reduces the validation to a couple of comparisons, greatly reducing the overhead introduced by incremental validation.

Global commit counter (GCC) developed by Spear et al. [SMSS06] is one early form of time-based STM designs. In GCC, a transaction does not perform a validation if no writes were committed in the entire system since the last object was opened. This scheme can avoid many unnecessary validations, leading to performance improvement for situations where writes happen rarely in the system. However, this strategy is very conservative since a write to a shared object does not affect a particular transaction's consistency if that object is never used in that transaction.

Dice et al.'s [DSS06] *transactional locking II* (TL II) algorithm is another time-based STM implementation. Their algorithm associates a timestamp with every

```
1 if READ_ONLY = TRUE then
2 |  if O_i.ts > T.ts then
3 |  |  ABORT(T);
4 else
5 |  if O_i.ts > T.ts then
6 |  |  ABORT(T);
7 |  else
8 |  |  T.O ← T.O ∪ O_i;
```

Figure 3.1 : TL II validation.

```
1 if O_i.ts > T.ts then
2 |  T.ts ← TSC;
3 |  VALIDATE(T);
4 T.O ← T.O ∪ O_i;
```

Figure 3.2 : Lazy snapshot validation.

shared object at the time of its modification. A transaction acquires its own timestamp once at the beginning of execution, and commits successfully if none of the objects opened during the execution has a timestamp newer than the timestamp of the transaction. A transaction aborts if it encounters an object with a newer timestamp. An advantage of this algorithm is that it can completely eliminate the bookkeeping overhead for read-only transactions since it never validates opened objects, making it attractive in situations where contention is low and most transactions commit successfully. Figure 3.1 shows the major steps of TL II's validation strategy.

Riegel et al.'s [RFF06] *lazy snapshot* algorithm maintains multiple versions for each shared object and uses the global timestamp information to guarantee that the view observed by a transaction is consistent. In their algorithm, a transaction

can choose the object version to satisfy consistency. When a transaction encounters an object with a newer time stamp, its validity range is extended by doing a full validation. Figure 3.2 shows the major steps of lazy snapshot's validation strategy.

For convenience, we explain the meaning of the variables used in Figure 3.1 and 3.2. T is a transaction. O_i is an object. $READ\_ONLY$ indicates if the transaction is a read-only transaction or not. $T.ts$ is the transaction's timestamp. It is used to reason about the ordering of the transaction and the objects. TS is the variable to save the timestamp to be written into the objects being updated. TSC is the timestamp counter. It is a shared integer. $T.O$ is a list keeping the objects that need to be verified when doing a validation. $VALIDATE(T)$ checks if the invisible reads and lazy writes have been modified or acquired by other transactions. If they are, transaction T is aborted, otherwise it proceeds forward.

Global commit counter has the least memory overhead since it only adds a shared commit counter. The sizes of objects are unchanged. However, it will perform a full validation every time there is a write in the entire system, even if that write does not affect the current transaction. These unnecessary validations can sometimes lead to a relatively poor performance when compared to the other two algorithms. Transactional locking II eliminates both validation and bookkeeping overhead, but it can sometimes be too conservative and abort transactions that could commit successfully. It has the additional memory requirement of adding a timestamp to every object. Lazy snapshot gives a transaction more opportunity to commit itself by performing validation when a new timestamp is encountered, but this is done with the overhead of maintaining and validating past reads.

In my thesis we refer to a variant of the lazy snapshot algorithm that only keeps a single version of an object. Compared to GCC, lazy snapshot eliminates a superset of

full validations. Compared with TL II, lazy snapshot extends a transaction's validity range when it encounters an object with a newer timestamp. Lazy snapshot needs bookkeeping of all its past reads for the purpose of validation. It can execute a transaction to a point closer to the commit than either GCC or TL II. However, this is not always beneficial: it potentially increases the amount of wasted work being performed by a transaction that is doomed to abort. The bookkeeping overhead of Lazy Snapshot is the same as for the GCC, and larger than TL II. The memory overhead of lazy snapshot is the same as TL II and larger than GCC.

The major difference of TL II and lazy snapshot is in deciding when a transaction should be aborted. When a transaction opens a new object, Transaction Locking II assumes aborting the current transaction benefits the system the best and aborts it right away, but lazy snapshot validates the read set and gives the transaction a further chance to commit.

Research has also explored language models and semantics of supporting transactional memory. Harris et al. [HMPJH05] proposes language extensions to support composable transactions. Their extension includes *retry* and *orElse*. *Retry* provides the programmer the ability to abort an transaction and start it over again. *orElse* provides the ability to write transactions with alternative operations. A transaction can execute in one form for the first time and switch to another form when it aborts.

3.1.2 STM Performance Optimizations by Relaxing Transaction Semantics

Other research considers improving STM performance improvement by relaxing some of the transactional properties and extending the tools programmers can use to encode program-specific knowledge in their transactional applications. Using these tech-

niques sacrifices some programmability for better performance. Since they are not strictly conforming to the TM requirements, programmers need to spend more time and effort on reasoning about correctness of their applications. These techniques include early release [HLMS03], open nesting [Mos05, MH05, NMAT+07], and lowering the overhead of shared memory access [CH04].

Early release is a technique first proposed by Herlihy et al. [HLMS03] to reduce contention in their DSTM system. Early release allows a transaction to release some transactional reads before the transaction commits. A transactional read, once released, does not conflict with other concurrent transactions or incur the associated validation overhead. This reduces the probability of aborting the transaction and improves the overall performance. Early release requires more programming effort because it lays the burden of ensuring there are no conflicts on the programmer. Besides the programming effort, transactions using early release are not composable, so this technique affects program modularity as well. For example, a transaction A verifies if a node exists in a sorted linked list. It can go through the list and early release every node it opens. Thus, when transaction A finishes, all reads are already released. If the information gathered in transaction A is used in a later nested transaction B, the later transaction can no longer validate if the condition is still true because all relevant nodes have been released.

Open nesting [Mos05, MH05, NMAT+07] is a technique to exploit a higher semantic level of concurrency than is defined at the physical memory access layer where TM systems usually reside. It allows transactions to commit even in the presence of a physical conflict not affecting the application's semantics. The system usually needs to support necessary virtual constructs (not necessarily locks) that can be mapped to the semantic level where the significant conflicts actually happen. A TM system

supporting open nesting works with these high level constructs rather than constructs associated with raw memory access. For example, consider a nested transaction composed of two transactions, each of which inserts a node in a list. If the application only requires that these two transactions either both complete or neither of them complete but does not care if another transaction intervenes in between, open nesting can be used to improve performance. Similar to early release, using open nesting requires deep understanding of the application's semantics and is more difficult than a pure transactional approach. Moreover, open nesting is not composable. First of all, open nested transactions are not normal transactions. They only have relaxed transactional support. Up to now, there is still no a clearly defined semantics for composed transactions based on open nested transactions. For example, what should the semantic be for a transaction D like *atomic*{*A; open_atomic*{*B; C;*}}. Should D still be open atomic or not. Also, in open nesting, a higher layer of concurrency is established to model the concurrency at a higher semantic level than the physical memory access layer. The concurrency control at this high semantic level layer is not guaranteed to have the support like Transactional Memory. User supplied handlers are used to provide sufficient guarantee for providing the open nesting semantics. These user supplied handlers can raise the possibilities of deadlock, etc. One consequence of the absence of system level support is the composability of open transactions is comprised. To compose a bigger transaction using an openly nested transaction, a programmer first needs to understand the semantics of the openly nested transaction; he also needs to understand how the high level abstract virtual constructs are operated in the open nested transaction to avoid problems such as deadlock. Ni et al. [NMAT+07] gives one example showcasing the possibility of causing deadlock when using open nested transactions. Comparatively, transactions optimized based on our

Fast Read Extension behaves the same as transactions and can be composed into bigger transactions when no new update transactions are added to the application.

SNAP is a low-overhead interface for shared memory access [CH04]. It provides functions to get, validate, and upgrade the snapshot of an object. A SNAP read, unlike a regular transactional read, does not involve any bookkeeping information. Instead, the read returns a snapshot of the object. SNAP validation can verify if a snapshot held by the program is still valid. A programmer can also upgrade a read snapshot to a write snapshot when needed. Memory accesses in SNAP mode do not suffer the transactional overhead of recording the reads and validating them. Using SNAP needs additional programming effort of making correctness guarantees at the application level and gives a programmer the flexibility to optimize some memory accesses. A programmer explicitly manages the way a memory location is accessed and the switch between transactional and non-transactional modes. SNAP access mode is not composable as well.

3.1.3 Time Base Designs

In time-based STM systems, the time base is one critical component. It is responsible for generating new timestamps. The design of the time base is closely related to the STM system's performance. Most existing time based STM systems use a single counter as the time base. Riegel, Fetzer and Felbel explored using an external or physical clock that can be accessed efficiently or using multiple synchronized physical clocks [RFF07]. They noticed the potential scalability problem of a centralized single counter time base. Their solutions use external perfectly synchronized physical clocks or imprecise synchronized physical clocks with a limited error. The limitation of their work is the need for external hardware support, which could limit its areas of

utilization.

3.1.4 Other Related STM Research

Lev, Moir and Nussbaum [LMN07] propose a phased transactional memory (PhTM). Their system supports switching between different "phases" that represents different forms of transactional memory support. The STM designed by Ananian and Rinard [AR05] supports strong isolation. The strong isolation is achieved by instrumenting each shared object with a sentinel value, so every shared read/write outside a transaction can verify this value. Manassiev, Mihailescu and Amza's DMV [MMA06] implements transactions over the Treadmark software distributed shared-memory system. Their system takes advantage of the memory model used in Treadmarks to propagate transactional changes and provide consistency guarantee.

3.2 Hardware Transactional Memory

Hardware transactional memory explores special-purpose hardware mechanisms to support transactional memory accesses. While HTM is not as flexible as STM, it is able to achieve better performance than STM systems.

Stone et al. designed the Oklahoma Update protocol [SSHT93] that supports atomic read-modify-write operations on a bounded number of memory locations. It aims to simplify creating concurrent operations on shared data structures. Herlihy and Moss's seminal ISCA 1993 paper [HM93] coined the term *transactional memory.* Their system augments a process with new instructions and a transactional cache to provide transactional semantics. Rajwar and Goodman's speculative lock elision (SLE) [RG01] is a hardware mechanism that supports optimistic execution of critical sections. The key observation of SLE is that instead of locking a location, it

can monitor the location and skip the locking and releasing. A processor only acquires a lock if repeated conflicts is detected for a location. Rajwar and Goodman later proposed transactional lock removal (TLR) [RG02] extending to SLE by using timestamp-based conflict resolution to support transactional semantics.

Transactional coherence and consistency (TCC) [HWC+04] by Hammond et al. is a shared memory model that all operations are executed inside transactions. In TCC, committing a transaction is arbitrated by a global token. In TCC, only one transaction can commit system-wide at a time. Ananian et al. designed an HTM called large transactional memory (LTM) [AAK+05] to support transactions with footprint larger than cache size by spilling cache lines to local memory. Moore et al. designed LogTM [MBM+06] that also allows a transaction to overflow transactional data from local data cache to memory in other part of the memory hierarchy. LogTM keeps a software log of memory locations updated in a transaction. These locations can be restored if an abort occurs. Their system does not allow a transaction to survive context switch.

Ceze et al. proposed an HTM implementation called *Bulk* [CTTC06] without leveraging the cache coherence to track transactional memory accesses. Instead, they compress the read and write sets of a transaction into signatures propagated to other concurrent transactions. The signature actually represents a superset of the read and write locations that leads to more false conflicts.

Ananian et al. later designed another HTM system - unbounded transactional memory (UTM) [AAK+05] that allows a transaction to survive context switches and hardware buffer overflow. In UTM, the memory states necessary for maintaining transactional semantics are stored in memory, decoupled from cache coherence layer.

Rajwar, Herlihy and Lai designed another HTM system called virtual transac-

tional memory (VTM) [RHL05]. VTM provides an abstraction layer for programmers to hide the complexities such as the limited hardware buffer size and scheduling duration. Zilles and Baugh's HTM system [ZB06] changes the deferred update of the VTM to direct update.

3.2.1 Hybrid Transactional Memory

Hybrid transactional memory uses hardware transactional memory as a fast path of transaction execution and uses software transactional memory as the backup path and slow path when the hardware transactional memory path is not feasible. Lie proposes a hybrid hardware-software transactional memory system (HSTM) [Lie04]. In his system, a transaction is first executed as a hardware transactional memory. If the execution in HTM is not successful, the transaction changes to software transactional memory execution. Kumar et al. designed a hybrid transactional memory system [KCJ+06] that first executes a transaction using hardware transactional memory. If the hardware execution fails, it executes the transaction in DSTM. So the system can use the performance advantage of HTM when feasible and falls back to STM for flexibility. Shriraman et al. also described Rochester transactional memory (RTM) [SMD+05]. RSM proposes to expose some hardware assisted instructions to software and let a transaction use them to accelerate the transaction's execution. RTM proposes to expose the control of which cache line should be or should not be transactionally monitored to software.

3.3 Atomic Constructs

In this section, I present the work related to multi-place isolation work.

As a method of synchronizing parallel applications, coarse-grain locking is easy to

use, but does not usually scale well. Fine-grain locking, on the other hand, delivers better parallel performance at the cost of being error-prone and much harder to implement correctly. Ad hoc implementations of non-blocking algorithms for parallel data structures are even harder to produce (a non-blocking algorithm for even simple data structures such as queues is a publishable result); however, they often achieve the best performance.

An alternative to creating ad hoc implementations for each data structure is to use a general purpose universal construction that allows them to be created mechanically. The idea is to separate the concept from the implementation: Ideally, one would prefer to use a construct that is as easy to use as course-grain locking, but that performs and scales just as well as fine-grain locking and ad hoc non-blocking implementations.

One such universal construction is `atomic`. It allows the programmer to specify a chunk of code that executes atomically: the effect of the execution is that either *all* or *none* of the changes made by the specified piece of code are visible to the outside world. No intermediate state of the computation should be observable outside of the *atomic* code.

3.3.1 Transactional Memory Implementation Approaches

The first atomic constructions are due to Herlihy [Her90]. His lock-free protocol consists of three steps. First, the thread uses the load-linked instruction to read a global pointer to the object to be updated. Next, the thread copies the entire object, validates the private copy, and applies sequential updates to it. Finally, it uses the store-conditional instruction to swing the global pointer to its local copy. If this succeeds, the operation is complete; otherwise, some other thread has succeeded and the thread needs to retry. Store-conditional can fail because of a hardware cache

conflict and other related reasons.

Herlihy and Moss [HM93, LR06] proposed a hardware scheme in which data structure operations are encapsulated into *transactions*, groups of memory updates that collectively represent the operation. A special transactional cache holds tentative updates to data as well as memory locations that have been transactionally read (but not updated). The cache coherence mechanism is then extended such that so long as the transactionally cached data are not invalidated, the transaction can continue. Once the updates are completed, a commit instruction releases them to other processors.

Chapter 4

Runtime Tuning of STM Validation Techniques

4.1 Introduction

Part of this chapter's work has been published in a paper that appeared in EPHAM 2008 [ZBS08b]. STM systems exhibit significant performance overhead over more traditional lock-based programming models. Significant improvements to the validation systems for STM have been published. Transactional locking II and lazy snapshot validation techniques have so far shown the best performance for a wide variety of applications. Unfortunately, the performance of these two techniques depends heavily on the application: the number of concurrent jobs, the length of the transaction, and the read/write ratio of the shared objects can significantly favor one technique over the other. Moreover, for long-running applications that change their behavior over time, neither of these two techniques is optimal. In this chapter, we present a runtime tuning strategy that uses profiling to determine the most profitable validation technique. Our runtime tuning strategy can behave as an arbitrary mix of transactional locking II and lazy snapshot techniques depending on the state of the STM system. We evaluated this technique on a set of STM benchmarks and show that our strategy performs within a couple of percent of the best validation strategy for a given static workload scenario, and that it outperforms both of the above techniques by up to 18% in long-running, dynamically-changing scenarios.

4.2 Adaptive Validation Selection

Global commit counter, transactional locking II, and lazy snapshot all reduce the overhead of the incremental validation by leveraging global time information. They effectively reduce the number of times the transactional system has to perform full object-level validations for transactions and thus improve the overall performance. But none of them can consistently outperform the other two in all scenarios. In this section we first present a threshold hybrid time-based validation technique and show its performance characteristics. Then we present a runtime tuning technique based on our observations of the performance space characteristics of our threshold hybrid technique.

4.2.1 Threshold Hybrid Validation Strategy

In lazy snapshot, a transaction can conduct several validations before its final commit. In transactional locking II, a transaction never performs a full validation for read-only transactions and only performs a full validation for update transactions that reach their commit phase. Our threshold hybrid validation consists of two phases. In the first phase it behaves as Lazy Snapshot: it performs a full validation every time it opens an object with a timestamp newer than the current transaction's *candidate linearization point* (CLP). It keeps a count of the number of open objects since the start of the transaction, and when this count reaches a certain threshold, it switches to the second phase. In the second phase, our strategy behaves as transaction locking II. It does not perform any more bookkeeping, and aborts if it encounters a new object.

The behavior of our threshold hybrid strategy is controlled by a simple threshold. If this threshold is 0, our hybrid strategy behaves as Transactional Locking II.

```
1 if O_i.ts > T.ts then
2 |  if T.oc < THRESHOLD then
3 |  |  T.ts ← TSC;
4 |  |  VALIDATE(T);
5 |  |  T.O ← T.O ∪ O_i;
6 |  else
7 |  |  ABORT(T);
```

Figure 4.1 : Threshold hybrid validation.

```
1 foreach O_i in T.O do
2 |  if O_i is different from its version in the shared memory then
3 |  |  ABORT(T);
```

Figure 4.2 : VALIDATE(T).

When the threshold is a large number, it behaves similar to Lazy Snapshot. For a threshold in between, it behaves as a arbitrary mix of transactional locking II and lazy snapshot. Figure 4.1 shows the major steps of our threshold hybrid validation strategy. Figure 4.2 shows the VALIDATE function used in Figure 4.1. *TSC* represents a timestamp counter. *T.ts* represents the timestamp of a transaction *T*. *T.O* is the set of transactional objects read by a transaction T. A transaction T verifies its consistency by checking if any of the transactional objects *O_i* in *T.O* is modified since *O_i* is read. *VALIDATE(T)* is the function of validating transaction T. *VALIDATION(T)* aborts T if it finds T is not consistent, as shown in Figure 4.2.

We have experimented with different heuristics for deciding when to switch to the second phase in our validation strategy. One method is to count all open objects and compare the count with the threshold on every full validation or on every object open. Another method is to count all full validations and compare that count to

the threshold before doing a full validation. Since we found little difference in the performance of these heuristics, in this chapter we assume a method that counts all open objects and does a comparison at every full validation.

One potential danger associated with time-based validation techniques is starvation. There is a possibility that a transaction keeps being aborted because it keeps observing an inconsistent state of the shared memory. This will happen more often in TL II than in lazy snapshot, since TL II aborts the transaction as soon as it opens a newer object. Our threshold hybrid validation naturally fits in between TL II and lazy snapshot as far as possibility of starvation is concerned.

4.2.2 Implementation

Our implementation is based on Rochester software transactional memory (RSTM) [MSH+06] release 2, which in general offers significant performance gains relative to its predecessors [SMSS06]. It supports both visible and invisible reads, and both eager and lazy acquires, and uses deferred updates. The experiments are executed on a SunFire 6800 cache coherent multi-processor machine, with gcc 4.1.2 compiler.

RSTM is a non-blocking C++ library built to support transactional memory in C++. It accesses an object through its header. The header has a clean bit to indicate whether the object is owned by a transaction. The object header points directly to the current version of the object. Each thread maintains a transaction descriptor that indicates the status of the thread's most recent transaction. The transaction descriptor also maintains lists of objects opened for read-only and read-write access. We extend RSTM by adding a timestamp field to the object header and adding a candidate linearization point field to the transaction descriptor. Figure 4.3 depicts

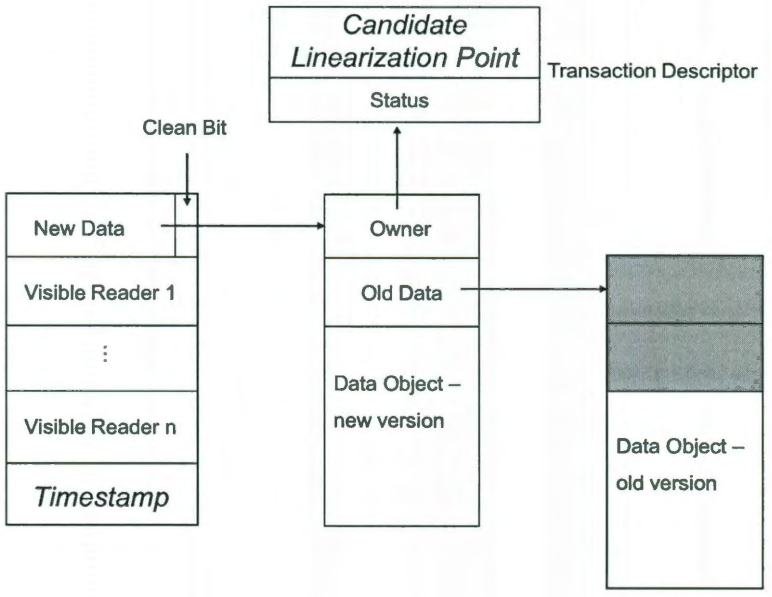


Figure 4.3 : RSTM metadata with addition for threshold-based validation.

our modification to the RSTM metadata.

The timestamp TS of each shared object indicates the time when the object is last modified. Closely related to timestamps are candidate linearization points. Each transaction keeps its own candidate linearization point which indicates the last time this transaction has observed a consistent state. This data allows the transaction to explicitly know where a potential linearization point lies relative to all the writes that have happened in the system. Having this knowledge enables the transaction to quickly decide to skip validation if it knows that the object it is trying to open has not been modified since its current CLP. When a transaction starts, its candidate linearization point is initialized to the beginning of the transaction. In lazy snapshot, CLP is updated each time a transaction opens a younger object and successfully validates past reads.

We use a global counter TSC to reflect the current time relative to the start of the program. TSC can be read concurrently by all the transactions in the system.

4.2.3 Benchmarks

We use six benchmarks in our experiments. They include a sorted linked list (LinkedList), a sorted linked list with hand-coded early release mechanism (LinkedListRelease), a red black tree (RBTree), a hash table (HashTable), a web cache simulation using the least-frequently-used page replacement policy (LFUCache), and an adjacency list-based undirected graph (RandomGraph). All of these benchmarks are taken from the RSTM release 2 distribution [MSH+06]; however, it should be noted that we have added a read-only lookup operation to RandomGraph.

LinkedList, LinkedListRelease, and RBTree contain values from 0 to 255. HashTable has 256 buckets with overflow chains. LFUCache tracks page access frequency in a simulated web cache using an array-based index and a priority queue. RandomGraph connects four randomly chosen neighbors with a newly inserted node. When any node is inserted or removed from the graph, the vertex set and the degree of every node is updated accordingly. The graph is implemented using a sorted list of nodes; each node has its own sorted list of neighbors. Transactions in RandomGraph exhibit a high probability of conflicting with each other.

To show the impact of different thresholds on performance, our experiments measure the throughput of the whole system when the threshold changes. To change the contention levels, we set up benchmarks with different numbers of competing threads (from 2 to 16). The lookup/insert/remove ratio per thread is 98%/1%/1%.

Figure 4.4 shows how the performance of our benchmarks changes with different thresholds. The x-axis of each figure is the threshold. The y-axis of each figure is the

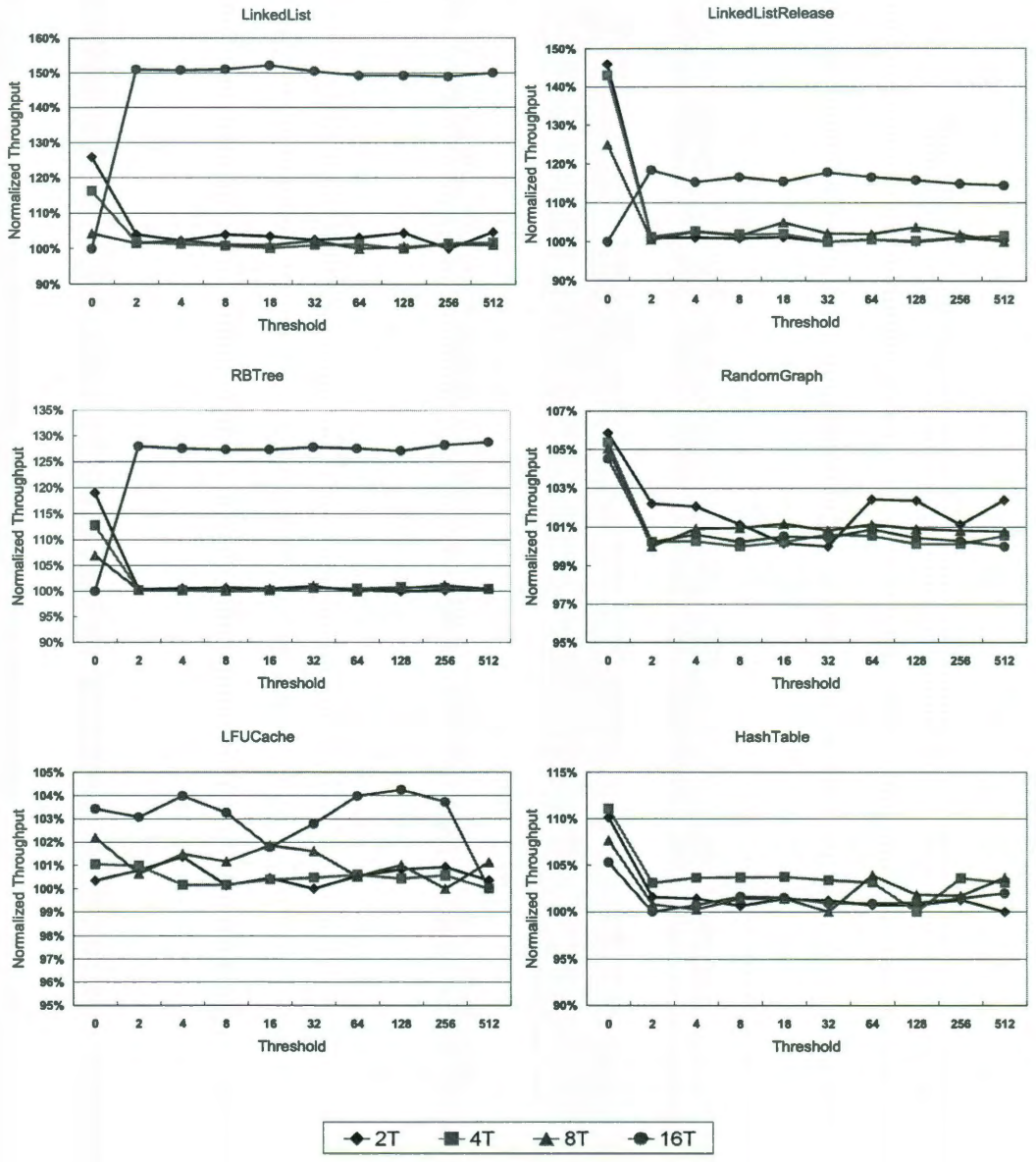


Figure 4.4 : Threshold hybrid strategy performance space.

throughput of transactions per second normalized to the smallest throughput with the same thread number. We use normalized throughput to show the shape of the

performance space of four different thread numbers of each benchmark. For each benchmark, we measured the cases of 2, 4, 8 and 16 threads, denoted with 2T, 4T, 8T and 16T in the legend.

The results show a nearly monotonically increasing or decreasing performance space most of the time across a variety of benchmarks and contention ratios. Notice that a STM system with a threshold of 0 is effectively TL II and LSS with a large threshold. Therefore in other words, the monotonic performance space suggests that if, for a given scenario, one of the validation techniques (TL II or lazy snapshot) performs better than the other, then it will also very often outperform our threshold hybrid technique. Since the performance increases or decreases as the threshold increases, the best performance is either achieved with TL II or lazy snapshot. The question is how and when to choose between the two validation techniques. This observation leads us to a coarse-grain runtime tuning strategy which switches the runtime validation strategy to the better one of transactional locking II and lazy snapshot. This runtime tuning strategy is described in the next section.

An interesting observation can be made about the threshold: it appears that in most of our benchmarks the "switching point" in performance happens around a threshold value of two. This is not very surprising after we observed that the average number of validations per transaction in all of our benchmarks is up to about 4. This also suggests that a different set of applications with a larger number of validations per transaction may create contention scenarios where the threshold hybrid strategy would outperform both of the extreme cases. Investigation into the practical usability of the threshold hybrid technique is a subject of our future research.

```
1 if ContentionChanged is TRUE then
2 |  T_old ← T_new;
3 |  T_new ← TestRun(Alternative Strategy);
4 |  if T_new < T_old * ratio then
5 |  |  Use New Strategy;
6 |  else
7 |  |  Use Old Strategy;
```

Figure 4.5 : Runtime tuning algorithm.

4.2.4 Runtime Tuning

Our runtime tuning strategy is based on the observation of the nearly monotonic property of the performance space. Our solution is to monitor the contention change in the system and select the right strategy when the change happens.

Figure 4.5 shows the major steps of our runtime tuning strategy. In Figure 4.5 T_{new} is the execution time of the new test run and is initialized to a very small number. T_{old} is the execution time of the previous test run. The STM system starts by running both transactional locking II and lazy snapshot for a short time and picking up the better one to continue. When the contention monitor detects a contention change, our runtime tuning system switches to the other strategy for a short time, then compares the performance with the current strategy. If the newly tested strategy is better, the system stays in that state, otherwise it switches back to the original strategy.

Our strategy has two major components - a contention monitor and a heuristic selector. The contention monitor keeps track of the runtime contention level and informs the runtime about any changes. The heuristic selector selects the best validation heuristic to continue the execution.

The contention monitor needs to be sensitive to contention level changes. We

use the wall clock time to commit a certain number of transactions as the indicator of the contention level. One advantage of using the wall time as the contention indicator is that it incurs very little overhead. Moreover, wall time is a relatively stable and accurate indicator to filter micro-variations in the TM system performance, which can vary up to 20% for short periods of time in our experiments, even for a constant contention level. In our implementation, the contention level monitor measures the accumulated sum of times it takes to commit 10,000 transactions (an arbitrary number). Depending on how fast the application changes its behavior, a larger or smaller number may be more suitable.

If the time spent on two consecutive sequence of transactions varies by more than some threshold, the monitor reports it as a contention change and informs the heuristic selector. In our experiments we set the threshold to 5%.

The goal of this chapter is to demonstrate that given a way of accurately monitoring the contention change, it is beneficial to use this information to guide the STM runtime to use the more suitable strategy. Designing an accurate contention monitoring scheme for general applications is beyond the scope of this chapter. We experimented with different strategies to estimate the contention level, including the ratio of total aborts over commits and recent aborts/commits ratio, but did not find a meaningful correlation between those indicators and the contention level.

4.3 Evaluation of Runtime Validation Tuning

We tested our runtime tuning strategy using RSTM 2 benchmarks and two benchmarks from the STAMP [CMCKO08] suite. The RSTM 2 benchmarks are relatively small benchmarks. The two benchmarks from STAMP are Labyrinth and Intruder that are relatively bigger applications.

We tested the threshold hybrid strategy and runtime tuning strategy on a SunFire 6800 computer with 16 UltraSPARC III processors running at 1.2 GHz using the RSTM 2 micro benchmarks. We ran each of these benchmarks with various number of threads, using the standard RSTM test driver. To test the effectiveness of the runtime tuning strategy on bigger applications, we measured the performance of Labyrinth and Intruder on a Sun Niagara 1 computer with 32 hardware threads using various number of threads.

4.3.1 Experiments based on RSTM 2 benchmarks

For RSTM 2 benchmarks, we evaluated our runtime tuning strategy in two scenarios. First, we compared our runtime tuning strategy against TL II and lazy snapshot in a dynamic scenario where the contention level changes throughout the execution of the program. Second, we compare our runtime tuning strategy against TL II and lazy snapshot in a scenario where the lookup/insert/remove ratio of all threads is constant throughout the whole run.

For the dynamically changing scenario, we set up the benchmarks to alternate between 2 thread execution and 16 thread execution (thus changing the contention level) every 5 seconds and measure the average throughput of 3 runs. Each run lasts 25 seconds. For all benchmarks except LFUCache, the lookup/insert/remove ratio of each thread is set to be 98%/1%/1%. The lookup/insert/remove is randomly generated following an uniform distribution. Figure 4.6 shows the performance results for the dynamically changing scenario.

Figure 4.7 shows the throughput (transactions per second) for the six different benchmarks, at different (but constant) contention levels, and different number of threads. The results are normalized to the performance of our threshold runtime

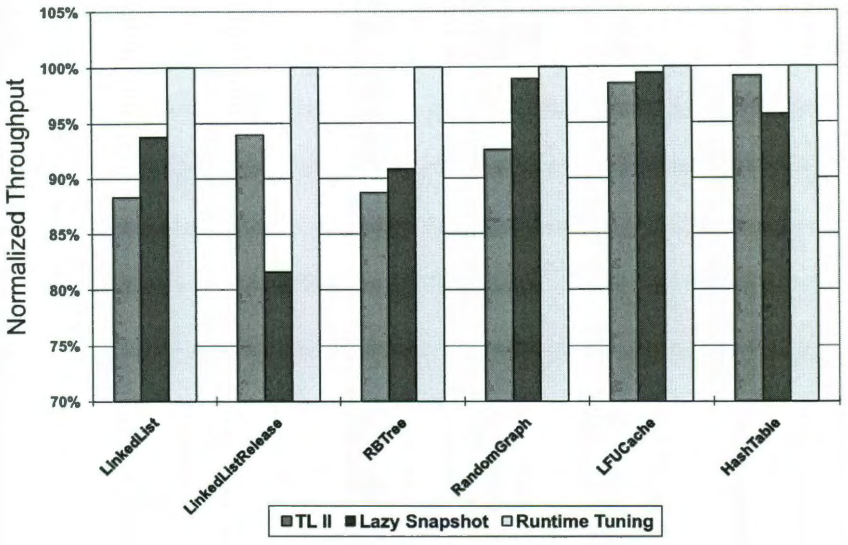


Figure 4.6 : Runtime tuning result. The throughputs are normalized to the throughput of the runtime tuning strategy.

tuning algorithm. For the constant contention case, we fix the thread number to be 2, 4, 8 and 16. We measure the average throughput of three runs. Each run takes 5 seconds. Since in the LFUCache benchmark, the read/write ratio cannot be changed, we only show how the result varies with a change in the number of threads.

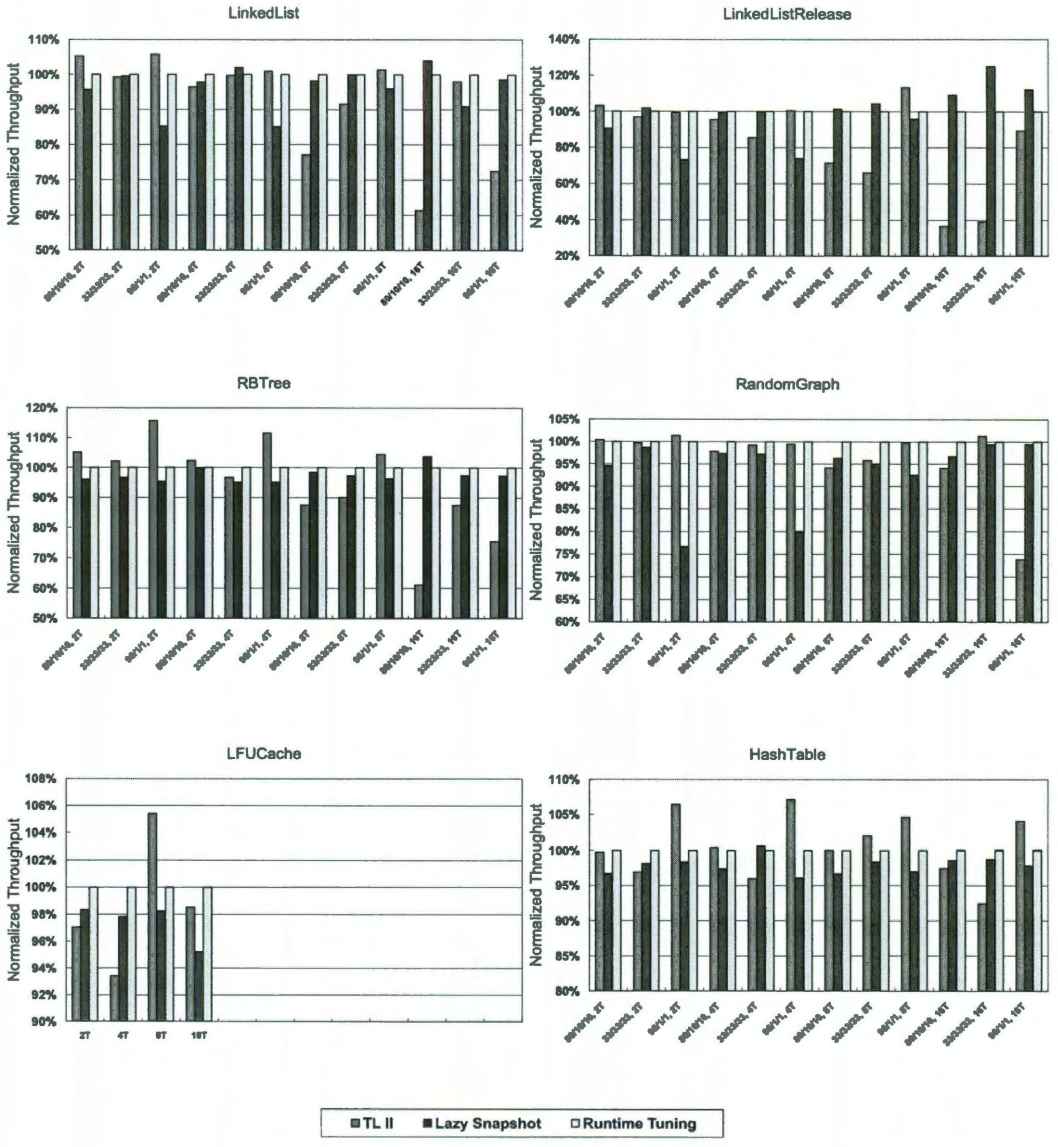


Figure 4.7 : Throughput for six different benchmarks, at constant contention levels, with different number of threads. Where applicable, the x-axes show setups of the experiments. The setup includes the mix of operations and the number of threads. For example, 80/10/10 means 80% lookups, 10% inserts, and 10% removes. 2T means using 2 threads. The throughputs shown are transactions per second and are normalized to the throughput using our runtime tuning strategy.

4.3.2 Experiments based on STAMP benchmarks

For Intruder and Labyrinth, we used the standard STAMP parameter as the setting of our experiments. The runtime parameter used for Intruder was "./intruder -a10 -l128 -n262144 -s1". The parameter used for Labyrinth was "./labyrinth -i inputs/random-x512-y512-z7-n512.txt". The STM system used in this part of experiments was a version based on TL II by adding the support for LSS and runtime tuning. We also modified both benchmarks by adding the runtime tuning code. The number of transactions to measure the average transaction time was set manually. For Intruder, this number was 10,000. For Labyrinth, this number was 50. For each benchmark, we collected the execution times for LSS, TL II and runtime tuning for 1 to 32 threads. Figure 4.8 shows the results for Intruder and Labyrinth. The group of columns represents the execution time of the benchmark for different thread numbers using different validation strategy. The two lines on each figure represent the speedup of the runtime tuning strategy against the slower or faster of LSS and TL II.

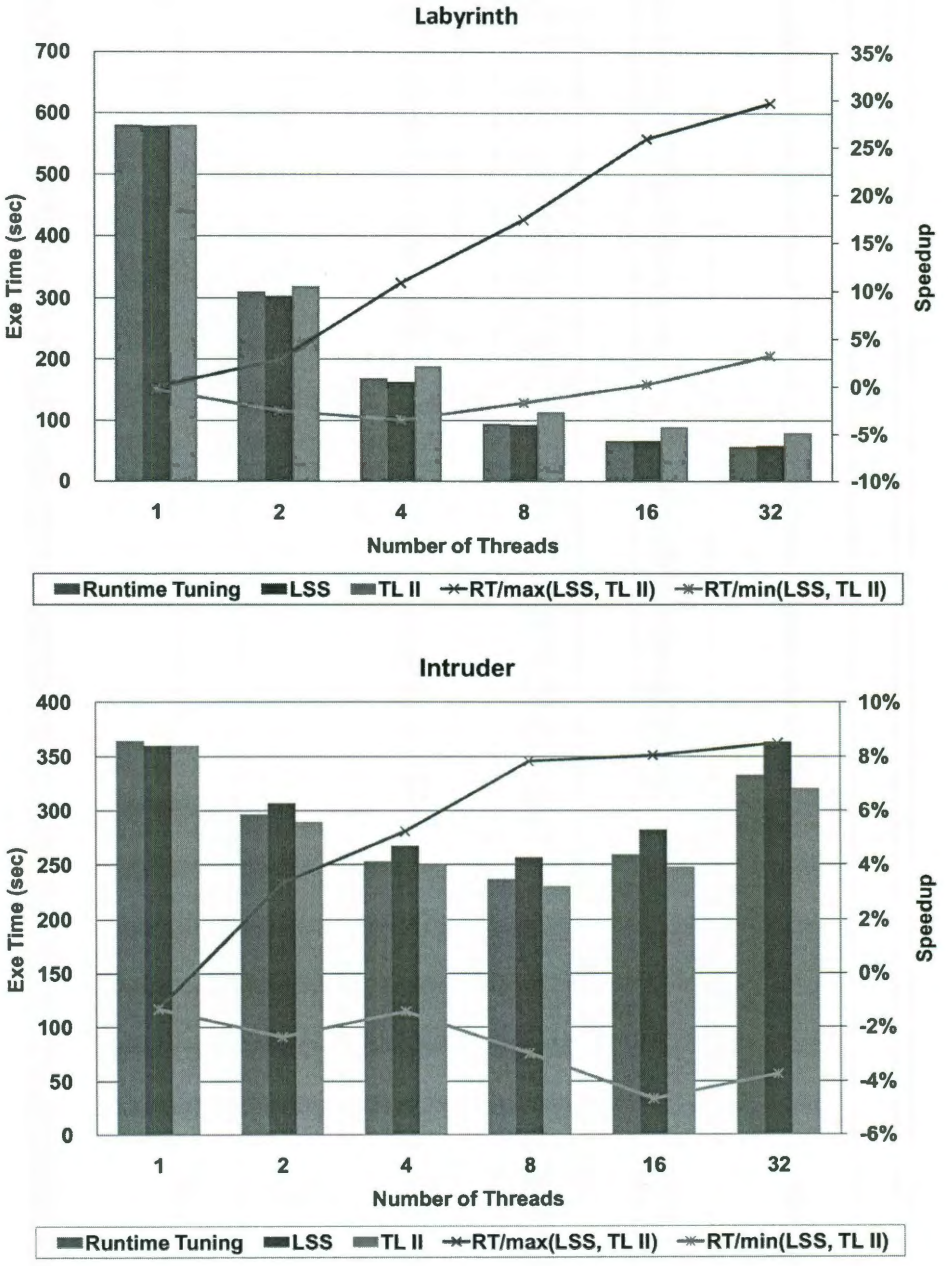


Figure 4.8 : Execution time and speedup for Labyrinth and Intruder. The bars represent execution times. The lines represent performance speedup of results using our runtime tuning strategy over TL II and LSS. The y-axis on the left is execution time. The y-axis on the right is speedup percentage.

4.3.3 Discussion

Figure 4.6 shows that our runtime tuning strategy outperforms both TL II and lazy snapshot in all cases when contention is dynamically changing. The performance improvement ranges from a couple of percent up to 18%.

We observe that under constant contention case Figure 4.7 shows that our strategy is very competitive with the best fixed strategy for any given scenario. This shows that a) the overhead of our strategy due to additional counters, run-time checks and occasional profiling of a sub-optimal strategy is relatively small, and b) that our strategy quickly converges to the behavior of the better fixed strategy. For constant contention, our strategy performs on average 15% better than the worst of the two, and is within 2% of the performance on average of the better strategy. Our strategy sometimes even outperforms both TL II and Lazy snapshot because our tuning can find finer-grain contention changes that happen even though the overall lookup/insert/remove ratio is statistically constant, because during short periods of time the contention level can still change.

The largest benefit of using a runtime tuning strategy presented in this chapter is that it prevents the system from using a clearly inferior heuristic for long periods of time. For example, for the LinkedList benchmark in the 16 thread case with 80%/10%/10% lookup/insert/remove ratio in Figure 4.7, if the system uses Transaction Locking II as the validation strategy, it will suffer a 43% performance degradation compared to the optimal strategy (lazy snapshot). For the same benchmark and the same lookup/insert/remove ratio, if the system is using Lazy Snapshot in the 2 thread case, it will suffer a 21% performance penalty compared to the optimal strategy (transactional locking II).

On the other hand, if the system is using our runtime tuning strategy, it will

perform at 6% below optimal in the worst case, and within 2% of optimal in 9 out of 12 cases for the LinkedList example. For an application that dynamically changes its behavior, using our tuning strategy can achieve overall performance that neither TL II or lazy snapshot can achieve.

For Intruder, its execution time when using TL II outperforms its execution time when using LSS. The performance difference between using TL II and LSS is up to 12% for 16 and 32 threads. Our runtime tuning strategy outperforms the slower one of TL II and LSS for most numbers of threads except for a single thread. The speedup is up to 8.5% for 32 threads. Compared with the faster of TL II and LSS, our runtime tuning strategy suffers a performance degradation of up to 4.7%. The results clearly show the runtime tuning strategy prevented Intruder from running at an inferior state for almost all thread numbers.

For Labyrinth, its execution time when using LSS outperforms its execution time when using TL II. The performance difference between using TL II and LSS is up to 27% for 32 threads. While the execution time with our runtime tuning strategy is tied with the best strategy for a single thread, it outperforms the slower of the two strategies for all other thread counts. The speedup is up to 29.7% for 32 threads. Compared with the faster one of TL II and LSS, the runtime tuning strategy results are mixed. The runtime tuning strategy outperforms the faster of TL II and LSS for some thread numbers and falls short for other numbers of threads. It outperforms by up to 3.2% and loses by up to 3.5%. The results clearly show the runtime tuning strategy avoided Labyrinth from running at an inferior state for almost all thread numbers. Also, the results show that the runtime tuning strategy outperforms the better of TL II and LSS for thread number 16 and 32. Better performance than both TL II and LSS clearly show the capability of our runtime tuning strategy to deal with

dynamic applications that a static strategy cannot.

The extra overhead of our runtime tuning strategy is very small. For the single thread case, the execution time using the runtime tuning strategy is almost the same as using TL II or LSS.

To conclude this chapter, our runtime tuning technique's performance is competitive to the state-of-the-art validation techniques and that it performs on par with the best heuristic for any given constant contention level. We have also shown that in a scenario with dynamically changing contention levels, our strategy consistently outperforms the state-of-the-art techniques by up to 30%.

Chapter 5

Commit Phase in Time-based STM

5.1 Existing Commit Phase Designs

The work in this chapter has been published in a paper that appeared in SPAA 2008 [ZBS08a]. Timestamp-based software transactional memory validation techniques use a global shared counter and timestamping of objects being written to reason about the sequencing of transactions and their linearization points thus reduces the number of object-level validations that have to be performed, which improves overall system performance.

5.1.1 Transactional Locking II (TL II)

TL II can save the overhead of bookkeeping for invisible reads for read-only transactions. TL II fixes its linearization point at the beginning of read-only transactions and does not validate.

TL II always performs a validation in its commit phase for update transactions. In TL II's commit phase, it forces an update of a shared integer counter and acquires a unique timestamp as shown in Figure 5.1. TL II also suggests one timestamp implementation to reduce the contention on the global shared integer counter by pairing the original timestamp with a thread ID for each shared object. In the commit phase of the latter version, TL II does not force an update of the shared integer counter as shown in Figure 5.2. The tuple of the timestamp and the thread ID is

```
1  if T.update then
2  |  ACQUIRE(T);
3  |  TS ← TSC;
4  |  while TS != CAS(&TSC, TS, TS+1) do
5  |  |  TS ← TSC;
6  |  TS ← TS + 1;
7  |  VALIDATE(T);
8  |  foreach O_i in T.O do
9  |  |  O_i.ts ← TS;
10 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.1 : TL II COMMIT(T) single counter version.

unique for each update transaction and satisfies the uniqueness requirement, allowing the commit phase to avoid the forced update of the shared counter. But in the commit phase of TL II, updates to the shared integer counter are unnecessary when validation fails. Both of the two timestamp designs have this problem as shown in Figures 5.1 and 5.2. The function ACQUIRE(T) acquires ownerships of all transactional objects that transaction T updates. Accesses of a transactional object whose ownership has been acquired by another transaction are detected as conflicts and need to be resolved by mechanisms such as aborting and backoff. Figures 5.3 and 5.4 show the TX_READ functions of TL II's single counter version and tuple version respectively.

To more directly compare different commit sequences, we changed TL II's *TX_READ* function to make it compatible with other versions i.e. LSS. This change consists of updating the transaction's timestamp and revalidating, instead of aborting, when an `TX_READ` detects that an object has been modified at a time later than the transaction's candidate linearization point.

```
1  if T.update then
2  |  ACQUIRE(T);
3  |  if T.ts = TSC then
4  |  |  TS_new ← CAS(&TSC, T.ts, T.ts + 1);
5  |  |  if TS_new != T.ts then
6  |  |  |  TS ← TS_new;
7  |  else
8  |  |  TS ← TSC;
9  |  VALIDATE(T);
10 |  foreach O_i in T.O do
11 |  |  O_i.ts ← TS;
12 |  |  O_i.id ← ID;
13 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.2 : TL II COMMIT(T) tuple version.

```
1  if m = write then
2  |  T.update ← true;
3  if O_i.ts > T.ts then
4  |  T.ts ← TSC;
5  |  VALIDATE(T);
6  T.O ← T.O ∪ O_i;
```

Figure 5.3 : TL II TX_READ(T, O_i, m) single counter version.

5.1.2 Lazy Snapshot (LSS)

LSS skips validation in the commit phase for read-only transactions as shown in Figure 5.5. It generates a unique timestamp using an atomic fetch-and-increment instruction for each update transaction, thereby serializing all update transactions when incrementing the shared counter. In our experiments we simulate the atomic fetch-and-increment instruction using CAS. LSS also skips validation when the timestamp counter is unchanged since its last validation: if no competing transaction has

```
1  if m = write then
2      T.update ← true;
3  if O_i.ts > T.ts or (O_i.ts = T.ts and O_i.id != ID) then
4      T.ts ← TSC;
5      VALIDATE(T);
6  T.O ← T.O ∪ O_i;
```

Figure 5.4 : TL II TX_READ(T, O_i, m) tuple version.

```
1  if T.update then
2      ACQUIRE(T);
3      TS ← TSC;
4      while TS != CAS(&TSC, TS, TS+1) do
5          TS ← TSC;
6      if TS != T.ts then
7          VALIDATE(T);
8      TS ← TS + 1;
9      foreach O_i in T.O do
10         O_i.ts ← TS;
11 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.5 : Lazy snapshot COMMIT(T).

committed, then no new conflicts can have been generated. However, if a final validation should fail, LSS will have unnecessarily updated the shared counter. LSS's TX_READ function is shown in Figure 5.6.

5.1.3 Global Commit Counter (GCC)

GCC is not a typical timestamp-based STM because it does not keep timestamps in shared objects. It uses a single counter as the time base. It skips the validation in the commit phase when there is no change to the counter since last time it was visited. It

```
1 if m = write then
2 |  T.update ← true;
3 if O_i.ts > T.ts then
4 |  T.ts ← TSC;
5 |  VALIDATE(T);
6 T.O ← T.O ∪ O_i;
```

Figure 5.6 : Lazy snapshot TX_READ(T, O_i, m).

```
1 if m = write then
2 |  T.update ← true;
3 if T.ts != TSC then
4 |  T.ts ← TSC;
5 |  VALIDATE(T);
6 T.O ← T.O ∪ O_i;
```

Figure 5.7 : GCC TX_READ(T, O_i, m).

forces an update of the shared counter (and ensures that the counter *does* get updated by the current transaction, in addition to any other updates that may happen to it at the same time by other transactions). It uses a unique timestamp for each update transaction. It performs a full validation when read-only transaction reads an object and detects a change to the global counter. The TX_READ and COMMIT functions are shown in Figures 5.7 and 5.8 respectively. As detailed in Section 5.2.4, this design admits a subtle race condition in which conflicting transactions may incorrectly commit together.

```
1 ACQUIRE(T);
2 if !TRY_COMMIT(T) then
3 |   VALIDATE(T);
4 |   TS ← TSC;
5 |   while TS = CAS(&TSC, TS, TS + 1) do
6 |   |   TS ← TSC;
7 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.8 : GCC COMMIT(T).

```
1 if T.update then
2 |   TS_new ← CAS(&TSC, T.ts, T.ts + 1);
3 |   if TS_new = T.ts then
4 |   |   result ← TRUE;
5 |   else
6 |   |   result ← FALSE;
7 else
8 |   if TSC = T.ts then
9 |   |   result ← TRUE;
10|   else
11|   |   result ← FALSE;
12 return result
```

Figure 5.9 : GCC TRY_COMMIT(T).

5.2 Commit Sequence Designs

Overall, the various timestamp designs in TL II, LSS, and GCC use two different methods to update the shared counter.

Forced update is used when the shared counter *has* to be updated. If used in a design where each update transaction uses a unique timestamp, all update transactions are serialized at updating the shared counter. This can become a bottleneck in highly parallel system where many transactions can reach their commit phase around

the same time.

Non-forced updates can be performed when an attempt to update the shared counter is sufficient to guarantee consistency. If the attempt succeeds (the CAS to update the shared counter succeeds), the commit sequence can continue as planned. If the attempt fails, a validation is needed to ensure consistency. The contention for the shared counter is greatly reduced in this case. TL II's suggested tuple implementation uses this scheme to update the counter part of the timestamp.

Another issue associated with existing designs is of unnecessary updates to the shared counter, which can force other transactions to perform an avoidable final validation at commit time.

In this section we present several alternatives to the commit sequences in the existing time-based STM systems. In particular, we relax the requirement for the *uniqueness* of timestamps and address the *unnecessary update* problem that can force transactions to perform unneeded validations (and by slowing their completion, increase the window for potential conflicts with other transactions). Unnecessary updates occur when a transaction leads to a shared counter update that does not correspond to the completion of any transaction.

5.2.1 Non-unique Timestamps

In prior timestamp-based STM systems, every write transaction needs to acquire a unique timestamp for the objects it updates. This uniqueness is achieved via a single integer timestamp or a tuple that has non-unique timestamp and a thread ID. Using a single unique integer requires an atomic increment to a shared timestamp counter for each update transaction, which serializes update transactions and reduces the scalability of the STM system. Using a tuple with thread ID reduces contention

because the thread ID is not shared; this avoids serializing all update transactions on the shared counter, but incurs additional space overhead.

Timestamp uniqueness is a sufficient condition for enforcing an order among transactions. However, as we demonstrate in this chapter, it is a more powerful property than strictly necessary, and leads to space and/or contention overheads. We present a method for relaxing the uniqueness requirement for timestamps and show that it can yield reduced contention on a shared counter without incurring the space overhead.

Our method is based on the observation that disjoint transactions can share a common timestamp if they commit concurrently. Within the commit sequence, an `ACQUIRE` operation ensures that any write/write inter-transactional conflicts are detected. Read/write conflicts, meanwhile, can be subsequently detected by a `VALIDATE` operation; so if two transactions have both successfully completed `ACQUIRE`, they may safely share their timestamp *provided* that they both validate their read set. Any moment in time between the `ACQUIRE` and the final commit can be used as the effective timestamp for a transaction. Although a transaction can still be aborted by a competitor after its `ACQUIRE`, this case is handled by the aborted transaction's status word update: when updating from `ACTIVE` to `COMMITTED` fails, the new versions of objects, with updated timestamps, are never committed as current. We argue more carefully the correctness of these claims later in this chapter.

Many authors have noted that, where possible, skipping validation during commit can improve transactional system throughput; this is done when the shared timestamp counter has not changed. When the counter *has* changed, we can choose either to abort and restart the transaction, or to force another update of the counter and validate the transaction.

5.2.2 Unnecessary Update Avoidance

We consider two scenarios for unnecessary updates in timestamp-based STMs that occur after the shared counter is updated as part of the commit sequence. The first scenario occurs when the transaction's own subsequent validation fails and it aborts. The second occurs when another transaction aborts (i.e., it detects a conflict) the one that has updated the timestamp. Both scenarios can potentially lead to redundant validation in otherwise uninvolved transactions when the STM system tries to skip a final commit-time validation. Since unnecessary update can potentially trigger a validation in each concurrently executing transaction, in a worst-case scenario the total amount of extra validation work performed can grow linearly with the number of transactional threads in a system — a clear scalability problem.

5.2.3 Commit Sequence Design Alternatives

The design space for commit sequences may be organized around several individual axes; we introduce nomenclature for them here.

We term *hard validation* the case where a `VALIDATE` operation is performed unconditionally in the commit sequence. In contrast, it may be skipped under certain circumstances in *soft validation*. If we always perform a validation, the update to the time base may be placed virtually anywhere in the commit sequence, and the expected value for the `CAS` may be either the candidate linearization point or a fresh read of the global counter. To skip commit-time validation (in cases where conditions are met for doing so), an update to the timestamp must occur before the validation attempt.

With a *hard increment*, the timestamp counter is always incremented during the commit sequence by the thread executing the transaction (typically via a looped

	validation	increment	TS acquire	thread ID	unnecessary update	side effect
V1	hard	soft	lazy	no	commit CAS fails	no
V2	soft	hard	eager	no	commit CAS fails	no
V3	hard	soft	eager	no	validation or commit CAS fails	yes
V4	soft	soft	eager	no	commit CAS fails	no
TL2T	hard	soft	lazy	yes	validation or commit CAS fails	yes
TL2C	hard	hard	lazy	no	validation or commit CAS fails	no
LSS	soft	hard	eager	no	validation or commit CAS fails	no

Table 5.1 : Commit sequence comparison.

CAS). With a *soft increment*, the timestamps can be shared, so the counter update can sometimes be skipped. *Soft increment* is done by identifying disjoint transactions and let them share the same timestamps. In some of our proposed variations to the commit sequence, it suffices for a committing transaction to observe that an update has been made to the shared counter, whether the transaction's executing thread was the one that successfully performed the increment or not. This avoids full serialization of all transactions and reduces contention on the timestamp counter, but it also leads to non-unique timestamps.

We characterize as *eager (or lazy) timestamp acquire* the case where the base value used to update the timestamp counter is read before (or after) the ACQUIRE operation. Finally, having *side effects* means that the TX_READ operation needs to

```
1 if m = write then
2 |  T.update ← true;
3 if O_i.ts > T.ts then
4 |  T.ts ← TSC;
5 |  VALIDATE(T);
6 T.O ← T.O ∪ O_i;
```

Figure 5.10 : V1, V2, V4 TX_READ(T, O_i, m).

perform validations not only when the object's timestamp is greater than transaction's timestamp, but also in some other cases. We summarize these differences in table 5.1.

Version 1 (V1): This design uses a single timestamp counter. It does not necessarily assign a unique timestamp for each transaction; instead it can share a timestamp across concurrently committing transactions. This allows it to perform only one CAS operation on the shared timestamp counter; even if the CAS fails, the counter is guaranteed to have been updated. Scalability is improved as contention on the counter is reduced, and the strict serialization of transactions on the counter is relaxed, as in LSS, GCC, and TL2C (TL II with a single integer as the timestamp). Unlike TL2T (TL II using a tuple of an integer and a thread ID as the timestamp), this version avoids using additional space for a thread ID. One disadvantage of this design is that validation must always be performed during the commit phase. The `TX_READ` and `COMMIT` functions of V1 are shown in Figures 5.10 and 5.11 separately.

Version 2 (V2): This design also uses a shared timestamp counter. It attempts to avoid unnecessary updates in the timestamp counter by validating past reads each time a `CAS` fails, getting a fresh read of the counter for each attempted update. The main disadvantage of this design is that every transaction must force an update to the timestamp counter, which increases contention.

```
1  if T.update then
2  |  ACQUIRE(T);
3  |  VALIDATE(T);
4  |  TS ← TSC;
5  |  if TS = TSC then
6  |  |  TS_new ← CAS(&TSC, TS, TS + 1);
7  |  |  if TS_new = TS then
8  |  |  |  TS ← TS + 1;
9  |  |  else
10 |  |  |  TS ← TS_new;
11 |  else
12 |  |  TS ← TSC;
13 |  foreach O_i in T.O do
14 |  |  O_i.ts ← TS;
15 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.11 : V1 COMMIT(T).

The `TX_READ` functions in V1 and V2 are identical. The `COMMIT` function of V2 is shown in Figure 5.12.

Version 3 (V3): This design is similar in most respects to V1. The key difference from V1 is in updating the shared timestamp counter: Where V1's `CAS` uses as its expected value a fresh read obtained after performing the `ACQUIRE` operation, V3's `CAS` uses the most recently read value. In our correctness proof for this design, we show that the `ACQUIRE` operation is a critical point; this difference forces V3 to perform additional validations on TX_READ when the object and transaction timestamps match. This version is suitable for systems with abundance of parallelism and where the transactions have a very good chance to successfully commit. The *CAS* in V3 has a high probability failing and the commit can take the quick path in the if statement. HashTable is one benchmark where V3's performance stands out. V3's

```
1  if T.update then
2  |  ACQUIRE(T);
3  |  TS ← T.ts + 1;
4  |  while T.ts != TSC or T.ts = CAS(&TSC, T.ts, TS) do
5  |  |  T.ts ← TSC;
6  |  |  VALIDATE(T);
7  |  |  TS ← T.ts + 1;
8  |  foreach O_i in T.O do
9  |  |  O_i.ts ← TS;
10 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.12 : V2 COMMIT(T).

```
1  if m = write then
2  |  T.update ← true;
3  if O_i.ts >= T.ts then
4  |  T.ts ← TSC;
5  |  VALIDATE(T);
6  T.O ← T.O ∪ O_i;
```

Figure 5.13 : V3 TX_READ(T, O_i, m).

performance matches TL II tuple version, without the additional space requirements. Due to the additional validations when opening a object with an equal timestamp, V3's performance suffers when the validation overhead is high, as demonstrated in LinkedList and RandomGraph. The `TX_READ` and `COMMIT` functions of V3 are shown in Figures 5.13 and 5.14 separately.

Version 4 (V4): Our fourth design combines the ability to share timestamps (and the corresponding reduction in shared counter contention) from V1 with the ability to skip validation in the commit sequence from V2. As such, it has potentially the lowest overhead of any of our commit sequences. Like V2, V4 shares a common

```
1  if T.update then
2  |  ACQUIRE(T);
3  |  if TSC = T.ts then
4  |  |  TS ← T.ts + 1;
5  |  |  TS_new ← CAS(&TSC, T.ts, TS);
6  |  |  if TS_new != T.ts then
7  |  |  |  TS ← TS_new;
8  |  else
9  |  |  TS ← TSC;
10 |  VALIDATE(T);
11 |  foreach O_i in T.O do
12 |  |  O_i.ts ← TS;
13 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.14 : V3 COMMIT(T).

TX_READ function with V1. The COMMIT function of V4 is shown in Figure 5.15.

5.2.4 Theoretical Considerations

The following lemma presents a key insight into sequencing of ACQUIRE and VALIDATE operations for timestamp-based validation techniques.

Lemma 1. *Suppose that at a moment in time t_3, a transaction T_1 opens an object O for reading that was written by a transaction T_2. Let t_1 be the moment in time that transaction T_2 successfully performed its ACQUIRE operation during its commit phase. If there was a moment in time t_2 at which transaction T_1 performs a successful validation, and if $t_3 > t_2 > t_1$, then T_1 does not need to validate its state at the moment t_3 when it opens the object O.*

Proof. If there are more than one successful validations performed by T_1 between moments t_1 and t_3, we will assume without a loss of generality that t_2 is the moment

```
1  if T.update then
2  |  ACQUIRE(T);
3  |  if T.ts != TSC then
4  |  |  T.ts ← TSC;
5  |  |  VALIDATE(T);
6  |  TS ← CAS(&TSC, T.ts, T.ts + 1);
7  |  if TS = T.ts then
8  |  |  TS ← TS + 1;
9  |  else
10 |  |  VALIDATE(T);
11 |  foreach O_i in T.O do
12 |  |  O_i.ts ← TS;
13 CAS(&TX_STATUS, ACTIVE, COMMITTED);
```

Figure 5.15 : V4 COMMIT(T).

when that happens for the first time. Let t_4 be the moment in time when T_1 opens the object O for the first time. There are three possible scenarios:

- $t_4 < t_1$: the *ACQUIRE* action performed by T_2 will fail, since it will attempt to obtain ownership of O, which is already open by T_1 for reading. The conditions that the lemma requires are not met.

- $t_1 < t_4 < t_2$: if T_2 hasn't committed yet, then the *TX_READ* action by T_1 in the moment t_4 will fail, since it will attempt to open an object that has been locked by T_2. If T_2 has committed at a moment t_5, then $t_5 < t_4 < t_2$ and validation at t_2 by T_1 will include O and ensure that all subsequent reads of O (including the one at t_3) is consistent with the state of the transaction.

- $t_3 \leq t_4 < t_2$: let t_5 be the time that T_2 commits. If $t_5 > t_4$ then the *TX_READ* at t_4 will fail. If $t_4 > t_5 > t_2$ then the success of the validation at t_2 guarantees that T_1 and T_2 do not share any objects at t_2, allowing T_1 to serialize after T_2

and use the object O written by T_2. If $t_5 < t_2$ then the validation at t_2 ensures that there are no conflicts between T_1 and T_2 and that T_1 can serialize after T_2, which allows T_1 to open O for reading without validation.

□

Informally, Lemma 1 states that for any two transactions T_1 and T_2, if T_1 performs a validation after T_2 performs the `ACQUIRE` command, then after the validation, T_1 can open any objects written by T_2 without validation.

At this point, we need to emphasize an important property of the timestamp-based validation techniques: every time a transaction updates its linearization point $T.ts$ to the current value of the shared global counter TSC, it also performs a full validation. This happens when the transaction tries to open an object with a timestamp $O.ts > T.ts$, and also at the beginning of the transaction. For simplicity in following explanation, we can assume that a call to $VALIDATE(T)$ is made at the beginning of each transaction as well, since this call would not do anything because there are no objects open by the transaction at that time. Therefore, every update to $T.ts$ is accompanied by a call to $VALIDATE(T)$. Also, without loss of generality, we will assume that the update to $T.ts$ and the full validation happen at the same single point in time t_{Tts}, at the end of the call to $VALIDATE(T)$.

Theorem 1. *Algorithm V1 satisfies the consistency requirement of transactional memory.*

Proof. Since $V1$ always performs a validation during the commit phase, we only need to prove that $V1$ does not lead to inconsistency when opening an object.

Without loss of generality, suppose O is the object written by transaction T_1 and read by transaction T_2. We prove that for all possible timestamps written into O,

there will not be an inconsistency in T_2. Let t_{AQ} be the time that V_1 performs its $ACQUIRE$ action on line 2.

Since any scenario where $O.ts > T_2.ts$ forces a validation on $TX\_READ(O)$, we only need to consider cases where $O.ts \leq T_2.ts$. Let t_{T2ts} be the point in time when the $T_2.ts$ gets updated to its current value. According to Lemma 1, to prove consistency, it is sufficient to prove that $t_{T2ts} > t_{AQ}$.

Let $t(line\ X)$ be the point in time when line X in function $V1\ COMMIT(T)$ gets executed. There are three possible places where the timestamp that gets written into O can be generated: line 8, line 10 and line 12 in function $V1\ COMMIT(T)$.

- $O.ts$ is generated at line 8: CAS at line 6 was successful. To obtain a value from TSC that is greater or equal to TS computed at line 8, it has to be $t_{T2ts} \geq t(line\ 6) \Rightarrow t_{T2ts} > t_{AQ}$.

- $O.ts$ is generated at line 10: CAS at line 6 failed $\Rightarrow$ someone else has updated TSC between $t(line\ 4)$ and $t(line\ 6)$. To obtain a value from TSC that is greater or equal to $TSNew$, it has to be $t_{T2ts} \geq t(line\ 4) \Rightarrow t_{T2ts} > t_{AQ}$.

- $O.ts$ is generated at line 12: the comparison at line 5 failed $\Rightarrow$ someone else has updated TSC between $t(line\ 4)$ and $t(line\ 5)$. To obtain a value from TSC that is greater or equal to TSC from line 12, it has to be $t_{T2ts} \geq t(line\ 4) \Rightarrow t_{T2ts} > t_{AQ}$.

□

Theorem 2. *Algorithm V2 satisfies the consistency requirement of transactional memory.*

Proof. The proof that opening an object when $O.ts \leq T.ts$ without a validation is identical to the proof of Theorem 1, with an exception that the possible places to generate the timestamp written into O are line 3 and line 7 in function $V2\ COMMIT(T)$.

- $O.ts$ is generated at line 3: the CAS at line 4 succeeds $\Rightarrow$ the only way for $T_2.ts$ to obtain a value of TSC that is greater or equal to TS at line 3 is if $t_{T2ts} \geq t(line\ 4) \Rightarrow t_{T2ts} > t_{AQ}$.

- $O.ts$ is generated at line 7: the only way to exit the loop at line 4 is if both $T.ts = TSC$ comparison and CAS instructions succeed. Therefore, TSC is incremented every time an update transaction commits $\Rightarrow$ every update transaction has its own unique timestamp written to the objects it updates. Since $O.ts \leq T_2.ts$: at the moment t_{T2ts}, T_1 has updated TSC which means it has finished its while loop $\Rightarrow t_{T2ts} \geq t(line\ 8) \Rightarrow t_{T2ts} > t_{AQ}$.

Since algorithm $V2$ does not always perform a validation during the commit phase, we also need to prove that it does not violate consistency when test on line 4 fails immediately and the body of the while loop never gets executed.

They only way for the *while* loop at line 4 to exit immediately is if both conditions on line 4 fail $\Rightarrow T_1.ts = TSC \Rightarrow TSC$ has not changed since $t_{T1ts} \Rightarrow$ no objects were updated in the whole system since t_{T1ts}. Since T_1 has already performed a validation at t_{T1ts}, no validation is necessary at time $t(line\ 4)$. □

Theorem 3. *Algorithm $V3$ satisfies the consistency requirement of transactional memory.*

Proof. Since algorithm $V3$ always performs a validation during the commit phase, we only need to prove that it does not violate consistency when opening an object with

a timestamp $O.ts < T.ts$. Note that Algorithm $V3$ forces a validation when opening an object even when $O.ts = T.ts$, which is not the case with Algorithms $V1$ and $V2$.

The proof that opening an object when $O.ts \leq T.ts$ without a validation is identical to the proof of Theorem 1, differing points being the possible places to generate the timestamp written into O are line 4, line 7 and line 9 in function $V3$ $COMMIT(T)$.

- $O.ts$ is generated at line 4: the CAS at line 5 succeeded. Since $T_2.ts > T_1.ts + 1$ $\Rightarrow t_{T2ts} > t(line\ 5) \Rightarrow t_{T2ts} > t_{AQ}$
- $O.ts$ is generated at line 7: the CAS at line 5 failed. Since $T_2.ts > TSNew \Rightarrow$ $t_{T2ts} > t(line\ 5) \Rightarrow t_{T2ts} > t_{AQ}$
- $O.ts$ is generated at line 9: since $T_2.ts > TSC$ at $t(line\ 9) \Rightarrow t_{T2ts} > t(line\ 9) \Rightarrow$ $t_{T2ts} > t_{AQ}$

We also note that testing for equality when opening an object is necessary. The reason is the assignment at line 9. It is possible that $t_{T2ts} < t_{AQ}$ and $T_2.ts = O.ts = TS$ (at line 9), which would allow $T1$ to modify the object O and commit, and also allow T_2 to open the modified O without validation if the equality was allowed in function $V3$ TX_READ. This could lead to a *write-read* conflict if T_2 had already opened before t_{T2ts} a different object O_2 that is also being written by T_1. □

Theorem 4. *Algorithm TL II counter version satisfies the consistency requirement of transactional memory.*

Proof. Algorithm *TL II counter version* (the commit phase of the transactional locking II validation strategy) is a more conservative version of the algorithm $V1$, which allows the transaction an opportunity to abort when the CAS operation fails,

if it has been invalidated in the meantime by some other transaction. The consistency proof is nearly identical to the proof of Theorem 1. □

Theorem 5. *Algorithm $V4$ satisfies the consistency requirement of transactional memory.*

Proof. The proof that opening an object when $O.ts \leq T.ts$ without a validation is similar to the proof of Theorem 1, with an exception that the possible places to generate the timestamp written into O are line 6 and line 8 in function $V4\ COMMIT(T)$.

- $O.ts$ is generated at line 6: the CAS at line 6 failed $\Rightarrow$ TSC has been changed (by some other thread) between $t(line\ 3)$ and $t(line\ 6)$ $\Rightarrow$ the only way for $T_2.ts$ to obtain a value of TSC that is greater or equal to TS at line 6 is if $t_{T2ts} \geq t(line\ 3) \Rightarrow t_{T2ts} > t_{AQ}$.

- $O.ts$ is generated at line 8: the CAS at line 6 was successful $\Rightarrow$ the only way for $T_2.ts$ to obtain a value of TSC that is greater or equal to TS at line 8 is if $t_{T2ts} \geq t(line\ 6) \Rightarrow t_{T2ts} > t_{AQ}$.

Since algorithm $V4$ does not always perform a validation during the commit phase, we also need to prove that it does not violate consistency when both tests on line 3 and line 7 succeed.

If the test on line 3 was successful then the TSC has not changed since $t_{T1ts} \Rightarrow$ no objects were updated in the whole system since t_{T1ts}. Since T_1 has already performed a validation at t_{T1ts}, no validation is necessary at time $t(line\ 3)$.

If the test on line 7 was successful then the CAS was successful as well $\Rightarrow$ no objects were updated in the whole system since t_{T1ts}. Since T_1 has already performed a validation at t_{T1ts}, no validation is necessary at time $t(line\ 7)$. □

The correctness proofs for *TL II tuple version* and *LSS* are done in a very similar manner as $V1$.

While studying the different sequences of operations in the commit phase, we have also made an interesting discovery: the most recent implementation of the global commit counter commit phase as distributed with RSTM does *not* guarantee consistency. We can show this with an example of sequence of events:

Suppose a transaction T_1 reads object O_1 at time t_1 and writes object O_2 at time t_2, while a transaction T_2 writes O_1 after t_1 and reads O_2 before t_2. These two transactions clearly conflict and it should not be allowed that both commit successfully.

Suppose that $T_1.ts$ before entering commit phase was 3. Suppose that T_1 fails at `TRY_COMMIT` because the global counter has been changed to 4 by another transaction unrelated to T_1 and T_2. Then, before T_2 performs the `ACQUIRE`, T_1 successfully validates since T_2 has not acquired O_1 yet. Then before T_1 commits, T_2 updates $T_2.ts$ to 4 (when reading some new object), performs the `ACQUIRE` operation and using `CAS` changes the global counter from 4 to 5 successfully. After this both T1 and T2 can commit successfully.

In short, if a commit phase for an update transaction attempts to avoid performing the last validation, the update to the shared counter has to happen *before* the last validation. One solution is to move lines 4 through 6 ahead of line 3, making the commit sequence for GCC similar to LSS.

5.3 Experimental Results

We tested our commit sequence designs using release 2 of the Rochester software transactional memory (RSTM) [MSH+06], which offers significant performance gains relative to its predecessor [SMSS06]. We evaluated each design alternative with mul-

tiple benchmarks and various numbers of threads.

We extend RSTM by adding a timestamp field to the header of a transactional object, and by adding a candidate linearization point field to the transaction descriptor. Figure 4.3 depicts our modifications to the RSTM metadata.

We use the following benchmarks that we previously described in section 4.2.3 in our experiments: a sorted linked list (LinkedList), a sorted linked list with hand-coded early release (LinkedListRelease), a red black tree (RBTree), an undirected graph (RandomGraph), and a hash table (HashTable). These benchmarks are part of the RSTM release 2 distribution; we have only added a read-only lookup operation to RandomGraph.

The RandomGraph benchmark consists of an adjacency list based implementation of a graph. Insert operations add a new node to the graph and connect randomly chosen neighbors to it; delete operations remove a single node from the graph. Transactions in RandomGraph exhibit a high probability of conflicting with one another.

HashTable implements a 256-bucket table that uses overflow chains. The red black tree and linked list benchmarks contains sets of integer values in the range of 0 to 255. LinkedListRelease uses early release to reduce contention for early list nodes.

5.3.1 Test Methodology

Our test platform is a SunFire 6800 server with 16 UltraSPARC III processors running at 1.2 GHz. We ran LinkedList, LinkedListRelease, RBTree, RandomGraph, and HashTableeach with various numbers of threads and (where applicable) operation mix ratios, using the standard RSTM 2.0 test driver. We tested with invisible reads, lazy object acquisition, and the Polka contention manager. For each benchmark, we evaluated two operation mixes: a balanced workload with a mix of one-third each of

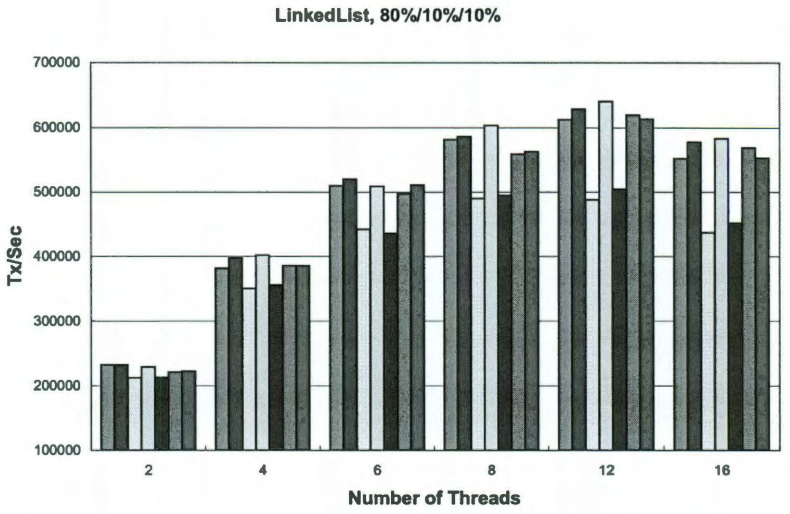


Figure 5.16 : LinkedList, 80% lookup/10% insert/10% remove.

lookup, insert, and remove operations; and a read-heavy workload with 80% lookups, 10% inserts, and 10% removes. We report results averaged across three five-second runs; spot checking confirms that using longer test runs does not noticeably alter the results.

5.3.2 Discussion

Figure 5.16 shows the results of **LinkedList**. Transactions in the LinkedList benchmark spend a large percentage of their execution time in validation, as the lookup, insert and remove functions all must traverse the list from the beginning to validate past reads and lazy writes. In LinkedList, versions 1, 2, 4, TL II counter, and LSS achieve better performance then version 3 and TL II tuple. We attribute the poor performance of version 3 to its need to validate when opening an object with a timestamp matching the transaction's current candidate linearization point; these extra

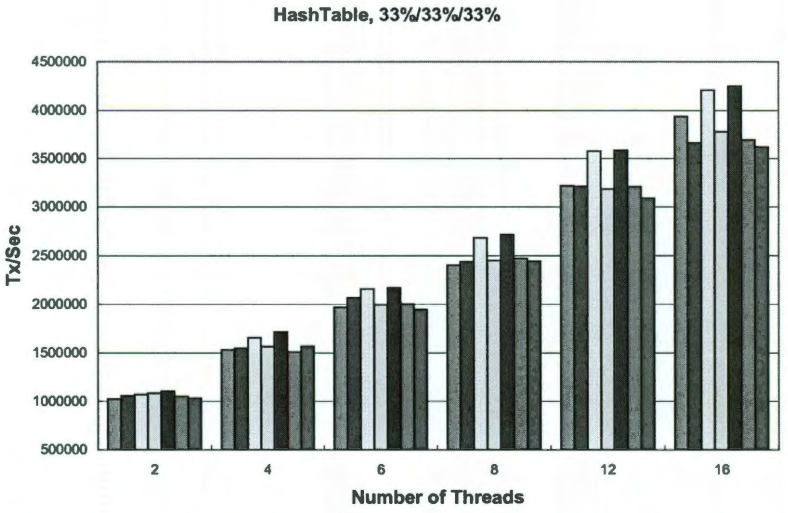


Figure 5.17 : HashTable, 33% lookup/33% insert/33% remove.

validations are not needed in versions 1, 2, 4, TL II counter version and LSS. TL II tuple version performs validations for objects with the same timestamps but different thread IDs; the extra validations manifest in the results as decreased throughput. The biggest performance difference we observe in LinkedList is 33% between V3 and V4 at 16 threads in the read heavy workload.

Figure 5.17 shows the results of **HashTable**. HashTable features a very large degree of parallelism, since transactions are usually short and disjoint. Transactions typically access only a small constant number of objects; hence validation is inexpensive, especially compared with benchmarks such as LinkedList. This leads to better performance from commit sequences that do not serialize all updates and that minimize runtime overhead.

We observe a difference of about 16% between TL II tuple version and TL II counter version at 16 threads in the balanced workload. We attribute the large per-

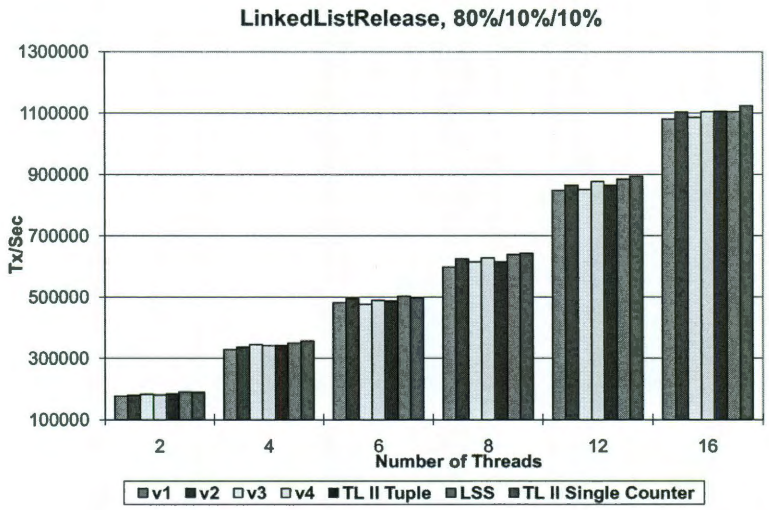


Figure 5.18 : LinkedListRelease, 80% lookup/10% insert/10% remove.

formance drop from TL II counter version to its serialization of transactions when updating the counter. Designs that reduce contention on the shared counter by allowing shared timestamps result in an improvement; the value of storing a thread ID stored with each object in TL II is clearly visible. Also allowing shared timestamps, version 3 nearly matches the performance of TL II; yet it does not incur the same space overhead from adding IDs to objects. Similarly, version 1 also uses non-unique timestamps. Version 2 and LSS update the time counter with every update transaction, a performance bottleneck at higher levels of concurrency.

Figure 5.18 shows the results of **LinkedListRelease**. LinkedListRelease uses early release to trim the working set size of transactions; the cost of validation reflects this smaller number of objects. This reduces the inherent cost of validation operations which in turn means that designs that attempt to eliminate a final validation in the commit sequence see little payoff. Further, the overall length of transactions is long enough that contention on the shared timestamp counter is very limited, even at high

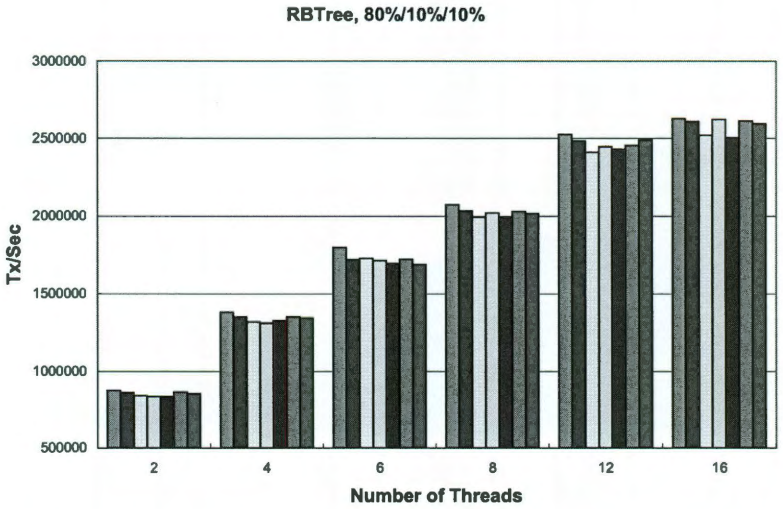


Figure 5.19 : RBTree, 80% lookup/10% insert/10% remove.

levels of concurrency. For these reasons, all designs give similar performance on this benchmark.

Figure 5.19 shows the results of **RBTree**. The tree structure of RBTree enables greater parallelism among transactions when they modify disjoint subtrees. Since the tree's red-black properties limit its maximum leaf depth to a logarithmic factor of the number of nodes, transactions are very short. We observe some performance loss at 16 threads with versions 3 and TL II tuple, due to their need for an additional validation when initially opening an object.

Figure 5.20 shows the results of **RandomGraph**. RandomGraph's behavior is similar to LinkedList: Its operations involve linear searches over adjacency lists in the graph data structure; hence, validation contributes a large part of its overhead. Further, its transactions have a high probability of conflict, and taking the time to perform a validation increase a window of opportunity in which another transaction

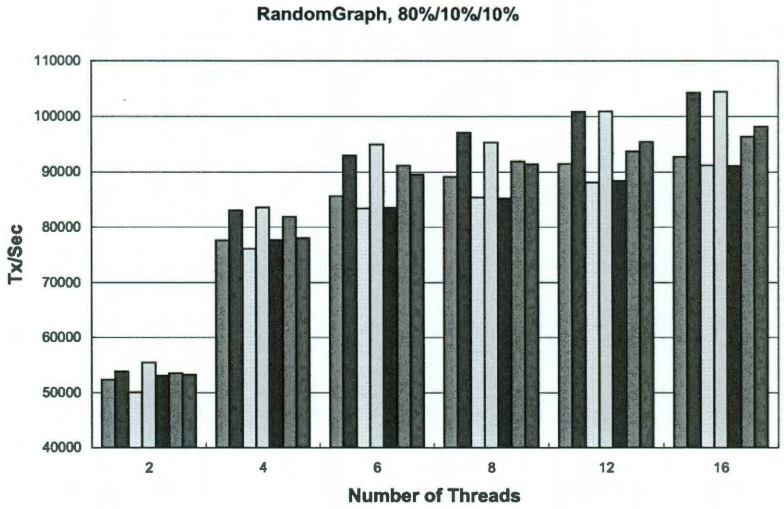


Figure 5.20 : RandomGraph, 80% lookup/10% insert/10% remove.

can discover conflict with one that is in the process of committing and abort it. So designs that seek to eliminate the final validation are particularly beneficial here. We observe around a 15% difference in throughput across levels of concurrency.

In summary, we have presented several variants of the commit sequence that either avoid forced updates to the shared global counter (and thereby reduce contention on it), or avoid performing validation in cases when it is safe to do so, or both. We have evaluated the proposed commit sequence variants on a set of transactional memory benchmarks, and shown variations that result in up to a 33% difference in overall system throughput. Our results show that the way how a commit phase is designed can significantly affect an STM system's performance and should be carefully considered when designing an STM system.

Chapter 6

Composability for Transactional Optimizations

In this chapter, I present a novel STM interface extension that enables composable application-specific transaction optimizations. This extension provides the necessary support to apply application-specific performance optimizations on existing transactions but without losing their composability.

6.1 Introduction

The work in this chapter has been previously published in TRANSACT 2010 [ZBS10]. Software transactional memory has made great advances towards acceptance into mainstream programming by promising a programming model that greatly reduces the complexity of writing concurrent programs. Unfortunately, the mechanisms in current STM implementations that enforce the fundamental properties of transactions — atomicity, consistency, and isolation — also introduce considerable performance overhead. This performance impact can be so significant that in practice, programmers are tempted to leverage their knowledge of a specific application to carefully bypass STM calls and instead access shared memory directly. While this technique can be very effective in improving performance, it breaks the consistency and isolation properties of transactions, which have to be handled manually by the programmer for the specific application. It also tends to break another desirable property of transactions: composability.

In this chapter, I identify the composability problem and propose two STM system extensions to provide transaction composability in the presence of direct shared memory reads by transactions. Our proposed extensions give the programmer a similar level of flexibility and performance when optimizing the STM application as the existing practices, while preserving composability. I evaluate my extensions on several benchmarks on a 16-way SMP. The results show that our extensions provide performance competitive with hand-optimized non-composable techniques, while still maintaining transactional composability.

I illustrate the composability problem itself in section 6.2. In section 6.3 I show my proposed interface extensions. Then I show the experimental results in section 6.4. Afterwards, I discuss the results in section 6.5.

6.2 Composability Problem

In this section, we identify the composability problem in the user's practice to improve STM application's performance by partially bypassing the TM system's consistency guarantee mechanisms. This type of user practice integrates application-specific knowledge into the code in order to recover performance lost due to the conservativeness of the STM system. For example, a programmer may choose not to use the general validation technique of the TM system, but to instead read directly from the shared memory and maintain necessary data structures by himself to ensure the correctness of the transaction. The programmer can still freely rely on the TM system for other guarantees such as atomicity and consistency of the transactional reads.

One motivating example to show this composability problem is the Labyrinth application in the STAMP [CMCKO08] benchmark.

The Labyrinth benchmark uses Lee's algorithm [Lee61] to find routes for a set of

```
1 begin atomic
2 |  copy global_matrix to local_matrix;
3 |  find a route in local_matrix;
4 |  if a route is found then
5 |  |  add the route found to global_matrix;
6 end
```

Figure 6.1 : Labyrinth transaction pseudocode.

sources and destinations in a three-dimensional matrix modeled after circuit board routing. For each pair of source and destination, the routing process expands its neighbor frontier till the destination is found. Then the routing process rewinds back to the source for a feasible route. The benchmark handles multiple pairs of sources and destinations in parallel. A straightforward pure TM implementation conducts the computation in the shared matrix, and generates many transactional reads along the search from the source to the destination. Given multiple concurrent threads, these reads have high probability to conflict with transactional writes and greatly decrease performance. In order to improve performance, the benchmark is implemented so that the matrix is first copied to a local array through non-transactional reads, and then all subsequent computation is done on the local array. The implementation employs a user-customized consistency checking algorithm. The consistency algorithm is used to write back routes found back to the shared memory. The customized consistency checking only needs to verify the nodes on the route and therefore greatly reduces the contention level and achieves much better performance.

Figure 6.1 is a pseudocode sketch of the algorithm used in Labyrinth. In the beginning of the transaction, a local matrix is created by copying from the global matrix. The read from the global matrix is a direct memory read without using TM

interface. When a route is written back to the shared memory, the program only writes it back when all cells on the route have not been changed in the meantime. The check of whether the cells are changed is done by the programmer.

This transaction is well formed to run concurrently with other transactions even though it reads from the global matrix non-transactionally. The transaction is atomic, consistent, and isolated within the application's context because the programmer carefully manages the consistency issues. But the composability problem arises when several of these transactions are merged into a larger transaction. For example, if the user wants to route a pair of sources and destinations in one transaction, he cannot use a nested transaction that has two of the routing transaction in Labyrinth. One of the problems is that in a delayed update TM model, the updates of the first transaction are delayed until the top level transaction commits. This makes the read of the array in the second transaction get the stale value and leads to incorrect results.

Below, we will take a look at a simple linked list example to illustrate the composability problem more clearly. Lists are frequently used in transactional applications. For example, both Genome and Intruder benchmarks from the STAMP benchmark suite [CMCKO08] use a linked list and it is reasonable to expect that the programmer would want to optimize the list transactions in ways similar to the one explained above. Unfortunately, list transactions are very often called within nested transactions. User optimizations would very likely break the composability of the transactions, making the code with nested transactions incorrect. This is exactly where the composability support described in this paper can help the programmer.

For example, imagine a data structure *Set* based on a sorted linked list that supports three transactions: insert, remove, and lookup, where the insert transaction incorporates two steps. The first step is to iterate through the list to find the lo-

```
atomic insert(List* list_ptr, int key) {
    Node* prev = TM_READ(list_ptr->head);
    Node* curr = TM_READ(prev->next);

    while(TM_READ(curr->key) < key) {
        prev = curr;
        curr = TM_READ(curr->next);
    }

    if (TM_READ(curr->key) != key) {
        Node* new_node = TM_MALLOC(sizeof(Node));

        new_node->next = curr;
        new_node->key = key;

        TM_WRITE(prev->next, new_entry);
    }
}
```

Figure 6.2 : Insert - pure TM version.

cation to insert the new node. The second step performs node insertion. A very straightforward approach, illustrated by the pseudocode in Figure 6.2, would be to use pure transactions and open every shared object transactionally.

While this is a very straightforward and simple way of creating such transactions, this approach does not scale very well with multiple threads. The main reason is that the nodes opened up to the insert point will incur many conflicts with other transactions, even though many of these conflicts can be shown to be benign. Knowing this, one of the ways to improve performance is to use the technique similar to the lazy concurrent list-based set algorithm developed by Heller et al. [HHL+05]. In the optimized version shown in Figure 6.3, each node is augmented with an extra field *marked* that indicates whether a node has been removed or not. The code searching

```
atomic insert_optimized(List* ptr, int key) {
    Node* prev = ptr->head;
    Node* curr = prev->next;

    while(curr->key < key) {
        prev = curr;
        curr = curr->next;
    }

    bool p_m = (bool_t)TM_READ(prev->marked);
    bool c_m = (bool_t)TM_READ(curr->marked);
    Node* next = TM_READ(prev->next);

    if (!p_m && !c_m && next == curr) {
        if (TM_READ(curr->key) != key) {
            Node* new_node = (Node*)TM_MALLOC(sizeof(Node));

            new_node->next = curr;
            new_node->marked = FALSE;
            new_node->key = key;
            TM_WRITE(prev->next, new_node);
        }
    } else {
        TM_RESTART();
    }
}
```

Figure 6.3 : Insert - uncomposable version.

for the location to insert a node reads the intermediate nodes directly (without going through the TM system calls). The correctness of the optimized version is ensured by validating that the two nodes neighboring the insertion point did not change during the insertion. The optimized version relies on the TM system to make sure the neighboring two nodes are consistently opened. This is done by transactionally opening the neighboring nodes.

```
1 begin atomic
2 |  insert(x);
3 |  insert(y);
4 end
```

Figure 6.4 : A composed list insertion example.

```
1 begin atomic
2 |  if lookup(value) == FALSE then
3 |  |  insert(x);
4 |  |_ insert(y);
5 end
```

Figure 6.5 : A composed conditional list insertion example.

As we will illustrate later in Section 6.4, the differences in overall application performance between accessing the data directly and accessing through the transactional interface can be very significant, suggesting that the optimized version is the way to go. Unfortunately, even though the insert transaction works well by itself within the context of the list based set, it is not composable. For example, the programmer may want to have two insert transactions in the nested transaction as in Figure 6.4.

Another example is that the programmer may want to insert a node to the set if the node does not already exist; and do this in a single transaction. The transaction will look similar to Figure 6.5.

In a pure TM implementation, the above two examples will work as expected. However, neither of the above nested transactions work correctly when the optimized versions of *insert* and *lookup* are used. There are two reasons that make the optimized version uncomposable.

The first problem arises in delayed-update STM systems [HLMS03, MSH+06].

```
1 begin atomic A
2 |  local_a ← global_a;
3 |  local_a++;
4 |  TM_WRITE(global_a, local_a);
5 end
6 begin atomic B
7 |  A;
8 |  A;
9 end
```

Figure 6.6 : A hidden update example.

In a delayed-update STM system, transactional writes are first made to a cached copy rather than into the shared variable. The cached copy is only committed to be visible to other threads when the transaction is committed. In the nested transaction, transactional writes in the first *insert* are not committed until the entire nested transaction commits. So the direct reads in the second *insert* read stale values of the transactional writes in the first *insert*. Consequently the computation of the nested transaction is no longer consistent. By comparison, in a pure TM version, all reads from the shared memory are transactional. Transactional reads respect the read-after-write dependencies across transactions and therefore do not incur this problem. We call this problem *the hidden update problem*, and it is tied to the implementation choice to use delayed updates within the STM. The hidden update problem does not occur in STM systems that use eager updates that immediately update transactional writes.

The hidden update problem can be seen more clearly in the following, even simpler example in Figure 6.6.

Transaction A reads the shared *global_a* variable and saves it to a local copy *local_a*

through a direct memory read. It then increments *local_a* and writes the new value to *global_a* using a transactional write. Note that reading *global_a* is not done through the TM interface and therefore does not suffer the associated performance overhead. Let us also assume that the application's semantics allows the transaction A to be implemented this way so that it does not interfere with other transactions in the application.

The hidden update problem arises when we nest two calls to transaction A in a nested transaction B. Suppose *global_a* is initialized to 0. The expected value of *global_a* is 2 after executing transaction B. But the second A's read of *global_a* reads the stale value 0; the final result of transaction B will be 1.

The second problem is a consistency issue. Direct reads do not force bookkeeping of any information in the transactional system. Therefore later transactions are not able to validate the consistency of previous optimized nested transactions. For example, consider the nested transaction from Figure 6.5 that is composed of one lookup transaction and one insert transaction. This transaction inserts node x only when *lookup(x)* indicates x is not already in the list. Note that all direct reads in *lookup(x)* are not recorded in the transaction and therefore cannot be validated later in *insert(x)*. *insert(x)* is dependent on the validity of the condition of *lookup(x)*, so if another concurrent transaction inserts x into the list after *lookup(x)* but before insert(x), the resulting list will have a duplicate element. The nested transaction is not able to discover this inconsistency. A pure TM implementation does not have this problem since all necessary information is recorded for all transactional reads and later nested transactions are able to validate the consistency of previous transactions.

To summarize, allowing optimizations that bypass TM system calls and access the shared memory directly breaks transactional composability because the read-

after-write and write-after-read dependences might not be respected properly when such transactions are nested.

Even though the read-after-write problem only occurs in the STM systems with a delayed write, we will address the composability for such systems as well, since delayed writes are frequently used in existing STM systems because they have the advantage of having a smaller conflict window and enable greater potential concurrency.

6.3 Fast Read Interface Extension

To meet both the need of optimizing the performance of STM applications using application-specific knowledge and of providing composability for optimized transactions so they can be more widely reused, we propose to extend the transactional memory programming interface with two additional operations. We introduce these two operations and their semantics next. We designed these extensions to maximize the extent to which programmers can benefit from optimizations embedded in the optimized transaction.

- *TxFastRead* encapsulates a fast read operation of shared memory and provides a hook for necessary operations to guarantee composability. It provides comparable performance compared to a raw read of a shared memory location. *TxFastRead* alone does not completely guarantee the consistency of fast reads; rather, it must work together with our *TxFlush* extension operation. *TxFastRead* should be used at every place where a direct shared read would have been used in an optimized transaction. *TxFastRead* does not employ the same heavyweight bookkeeping as does a regular transactional read, but adapts to execution context instead. It incurs no performance penalty in non-nested

contexts, yet provides composability when nested. Full details of this operation may be found later in this section.

- *TxFlush* is the counterpart operation for *TxFastRead*. First, it ensures that read-after-write dependences are respected. Second, it provides the necessary operations to guarantee consistency. *TxFlush* should be placed at places where these properties might be broken; typically, this is immediately before and after an optimized sub-transaction.

6.3.1 Composability Mechanisms

We experimented with two mechanisms for ensuring composability; we call them *lookup scheme* and *partial commit scheme*. With both, fast path *TxFastReads* return the shared value without any additional bookkeeping or validation. On the slow path, *TxFastRead* does perform some bookkeeping and validation. We carefully designed these two schemes so that the optimizations applied by the programmer can be preserved as much as possible. Within the optimized transaction, even when it is in a nested transaction, the TM system validation does not validate the fast reads. These fast reads are only validated when they are merged into the transactional read set. Therefore we expect less conflicts in the optimized transactions even when used as nested transactions.

Lookup scheme: Our lookup scheme solves read-after-write dependences by searching previous transactional writes. It ensures read consistency by recording the transaction reads in a fast read set.

This scheme does not commit transactional writes before the entire nested transaction commits. Therefore transactional writes are not committed to the shared memory when *TxFlush* is called. In this scheme, *TxFastRead* either looks up the

```
   input: An address addr
 1 begin TxFastRead
 2 |   if not nested then
 3 |   |   return *addr;
 4 |   else if addr is in write set then
 5 |   |   return value in write set;
 6 |   else if validate succeeds then
 7 |   |   record in fast read set;
 8 |   |   return *addr;
 9 |   else
10 |   |   abort;
11 end
```

Figure 6.7 : Fast read in lookup scheme.

```
1 begin TxFlush
2 |   merge fast read set to read set;
3 |   clear fast read set;
4 end
```

Figure 6.8 : Flush in the lookup scheme.

address from within the write list or reads directly from shared memory. In particular, in the context of a nested transaction, *TxFastRead* first searches the write list; however, in a non-nested context, this is skipped and the read is performed directly from shared memory. If the transaction is nested and the address is not found in the write set, then the transactional read set need to be validated (note that the fast read set is not validated here).

Figure 6.7 shows pseudocode for the *TxFastRead* function in lookup scheme.

In lookup mode, *TxFlush* merges the fast read set to the read set. Following nested transactions can validate the consistency of the enclosing transaction by

```
   input: An address addr
 1 begin TxFastRead
 2 |  if not nested then
 3 |  |  return *addr;
 4 |  else if addr is locked then
 5 |  |  clean up partial commits;
 6 |  |  abort;
 7 |  else if validate succeeds then
 8 |  |  record in fast read set;
 9 |  |  return *addr;
10 |  else
11 |  |  clean up partial commits;
12 |  |  abort;
13 end
```

Figure 6.9 : Fast read in PCM scheme.

validating only the read set. This addresses the inconsistency problem in nested transactions. Figure 6.8 shows pseudocode for the $TxFlush$ operation in the lookup scheme.

Partial commit scheme: Our partial commit scheme (PCM) solves the read-after-write dependencies by eagerly updating shared memory when a nested transaction commits. It solves the inconsistent read problem by recording fast reads.

In PCM, if the transaction is not nested, the shared value is returned directly. Otherwise, fast-read operations first check if the address to be read is locked. If it is locked, the transaction either aborts or waits for a while; otherwise, the fast read performs necessary validation and returns the shared data or aborts. In the partial commit scheme, the read does not search the write set. This is particularly useful when the write set is large and there are many variables in the write set. Note that the fast read is not a transactional read because it does not provide a mechanism

```
1 begin TxFlush
2 |   merge fast read set with read set;
3 |   clear fast read set;
4 |   commit current transactional writes;
5 end
```

Figure 6.10 : Flush in PCM scheme.

to guarantee consistency with other reads. The flush operation performs a partial commit that commits all pending writes to shared memory and locks them. It also merges the fast read set to the transactional read set. The nested transactions also needs to save necessary information to clean up the committed writes if the entire transaction fails. Figure 6.9 shows pseudocode for the $TxFastRead$ operation in the partial commit scheme.

In PCM, $TxFlush$ not only merges fast read set into the read set, but it also commits the write set to shared memory. The transaction holds locks for the committed writes, so accesses from other threads will detect a conflict and abort. Figure 6.10 shows pseudocode for $TxFlush$ operation in the partial commit scheme.

Both schemes have their advantages and drawbacks. On the fast path (when the transaction is not nested), both schemes perform similarly. They both perform a direct read of the shared memory and return that value. The difference is in their slow path. The lookup scheme's slow path needs to search the write set for every fast read so it suffers an associated performance penalty. The PCM scheme's slow path does not search the write set and therefore can be faster here especially when a large write set needs to be searched. In the lookup scheme, transactional writes are only locked when the enclosing transaction commits and therefore the locks are held for a shorter period of time. This can reduce the contention over the locks. In the

partial commit scheme, transactional writes are committed in steps when each nested transaction commits. Therefore the locks of early transactions are held for a longer time compared to lookup mode. This can lead to higher contention on the locks.

TxFlush can be exposed to the programmer or inserted by a compiler after every nested transaction. While exposing it may allow the programmer to identify the cases where *TxFlush* is not necessary and further optimize the performance, it could also increase complexity of the code and the possibility of writing erroneous code.

6.3.2 Composability vs. Reuse

We would like to point out here that the techniques we have described in this paper do not guarantee full reuse of the transactions, only composability. Composability is only a part of the reuse, even though a very important one. While this can be seen as a limitation of our approach, we want to point out that hand-optimized transaction code that bypasses the TM interface also cannot be reused in general, within a nested transaction or otherwise. Therefore our original claim that we provide composability for optimized code that bypasses the pure TM interface still stands.

To illustrate the reuse issue, let us consider the sorted linked list described earlier in section 6.2. It supports three transactions: lookup, insert, and remove. Suppose the programmer wants to add a new *increment* transaction that increments a value of a node by 1 (swapping it with the next node if necessary to preserve the order). Even if this new transaction is implemented using the pure TM interface without any optimizations, it will still break the existing hand-optimized code: adding a transaction that changes a node's value breaks the assumptions made in the original hand-optimized code. Namely, that only insert, remove and lookup can be performed on the list, which made the optimization possible. This is true even if the programmer

has used our extended TM interface to implement the optimizations. The programmer would have to revisit the assumptions made about the whole application and re-implement the optimized transactions with the new *increment* transaction in mind.

However, we also note that our TM interface extension *does* allow for the *increment* transaction to be implemented by simply making it a composition of a remove and an insert transaction. No changes to the existing code would be necessary.

Our extension still enables a significant amount of reuse, with a restriction that the new code does not contain transactional writes. If the added transactions only perform transactional reads and/or call the existing transactions, everything will perform as expected. Otherwise, a programmer will have to revisit the assumptions about the whole application and re-implement the optimized transactions.

6.3.3 Composability Guarantees

As discussed in Section 6.2, the composability problem can arise due to two issues — ignored read-after-write dependence and consistency violations.

The argument for general composability of transactions in our scheme is based on the following theorem.

Theorem 6. *Optimized transactions using our fast read interface are composable with themselves and transactional reads.*

Proof. We define S as the set that contains all transactions serializable with each other. S represents transactions available to compose new nested transactions. S includes both transactions that strictly follow TM requirements and transactions using our fast read interface. We define the transactions that use our fast read interface as optimized transactions in this proof. S satisfies the following property: transactions within any subset of set S are serializable. For set S, we claim that for

any nested transaction C that is composed of transactions from S, transactional reads, and local reads/writes, C is serializable with any transaction in S and transactions composed in a similar way to C.

First, we prove that the consistency of any sub-transaction A within a nested transaction N is still maintained in execution after A when executing N. This is to say that a sub-transaction remains consistent even when the execution moves out of its scope in the nested transaction. Second, we prove that any transaction using fast reads in S remains serializable when it is used in a nested transaction and running with only transactions either in S or composed in a way similar to itself.

The first part can be proved inductively from the four claims below for two transactions that are nested within a larger one. When composing new transactions, transactional reads in the new code containing calls to the nested transactions can be considered as simple and very small transactions. Transactional writes are not allowed in the new code, as discussed in the previous subsection. Following the strict nested transaction semantics, larger transactions containing more than two nested transactions can be rewritten with deeper nesting so that every level of nesting contains only two transactions. There is (conceptually) a $TxFlush$ at the beginning and at the end of every transaction that is optimized by using fast reads.

1. *Pure transaction followed by a pure transaction* does not violate consistency or read after write dependencies. This is trivially true in all STM systems.

2. *Pure transaction followed by an optimized transaction.* The reads in the pure transaction are recorded in the read set, which is validated later in the optimized transaction on every fast read. The writes within the pure transaction will be kept in the write set in the lookup scheme, which will be validated (in both

schemes) against fast reads in the following optimized transaction. The writes from the pure transaction will be committed to shared memory in the partial commit scheme during the $TxFlush$ between the two transactions, so they will be locked to the access from other threads, but can be accessed (either through fast reads or through transactional reads) from the following optimized transaction.

3. *Optimized transaction followed by a pure transaction.* In both schemes, the $TxFlush$ between the transactions will merge the fast reads from the optimized transaction into the pure transactional read set, so the pure transaction will validate all memory accesses against those reads. The writes in both the optimized and the pure transaction are transactional so they will be either kept in the write set or partially committed in the $TxFlush$ operation and validated accordingly in the pure transaction.

4. *Optimized transaction followed by an optimized transaction.* The fast reads from the first optimized transaction will be merged into the pure read set during the $TxFlush$ operation, so they will be validated against during the fast reads within the second optimized transaction. Transaction writes can be shown to be correct similar to the 2) and 3) above.

Second, we prove that any transaction T1 composed with transactions in S and local operations is serializable with any transaction T2 in S.

Since all writes to the shared memory in T1 are transactional writes, the atomicity property is automatically held for nested transactions. Because $TxFlush$ is inserted at appropriate places, there is no problem of guaranteeing consistency across sub-transaction boundaries. So we need to prove that if there is one optimized sub-

transaction $T1_1$ in T1, $T1_1$'s consistency still holds in the nested case. We prove this by contradiction. Suppose there exists one consistency checking method M that fails to catch some consistency violation in the nested case but not in individual case, then there must be at lease one consistency violation that is not detected in the nested case. If a consistency violation C escapes the detection in the nested case, we show that C is also able to escape the detection when the transaction runs alone, which conflicts with the assumption. For the case that C escapes, C can only escape due to the user consistency checking function because changing all fast reads to transactional reads will restore the consistency and the user consistency checking function is fully responsible for catching all cases violating the consistency. Similarly, the undetected violation can only be within some fast reads inside the optimized transaction because all other shared reads are transactional and their consistency is guaranteed by the TM system. For any violation C, C only depends on the reads within its enclosing optimized transaction, say T, given all previous reads are validated transactionally. We show that we can create a similar execution scenario that T is a running as a standalone transaction and raises the same consistency violation. A way of creating such an execution scenario is to replicate the execution sequence of the composed case using the same set of sub-transactions but in an uncomposed manner. This simulation is possible due to the fact that uncomposed version is less restrictive and can perform all interleavings that a composed version can.

To illustrate that any execution scenario of nested transactions can be simulated by using non-nested transactions, we use A, B and C. The two possible nested scenarios are A, B, C and C, A, B. We can use A, B, C and C, A, B to simulate the same execution using non-nested transactions. The claim can be proved inductively.

We have shown that an optimized transaction does not raise extra consistency

violation in the later part of execution of a nested transaction. Also it is obvious that in a nested transaction a sub-transaction's consistency will not be violated if it is followed by an optimized transaction. We have also shown that the consistency of an optimized transaction itself is not violated in a nested transaction composed in the way we specified earlier. Therefore we can say an optimized transaction is composable.

□

6.4 Experimental Results

We conducted our experiments on six benchmarks: Labyrinth, Genome, Intruder, Vote, sorted linked list, and nested sorted linked list. The experiments are performed on a 16-core SMP machine with four quad-core Intel Xeon CPU E7330 running at 2.40GHz. Three of the benchmarks in our experiments - Labyrinth, Genome, and Intruder - are from the STAMP [CMCKO08] benchmark suite. The other three were developed previously by the authors [ZBS08a].

We have (where applicable) four versions of each benchmark — pure TM, uncomposable, lookup scheme, partial commit scheme. We refer to the version that strictly follow TM requirements as the pure TM version. In the pure TM version, every read from or write to the shared memory passes through the standard TM interface. The pure TM version enjoys all the benefits from the TM system and requires the least effort to develop. We refer to the version using direct shared memory reads as the uncomposable version. The programmer is responsible for ensuring correctness through implementing isolation and consistency by hand. This version requires much more programmer effort. The versions that use our proposed fast read interface include the lookup scheme and partial commit scheme. In both schemes, the direct

accesses in the uncomposable version were replaced with the calls to the fast read interface. Similar to the uncomposable version, the programmer needs to guarantee the correctness. But unlike the uncomposable version, the transactions using our fast read interface can still be composed into larger transactions.

We implemented our extension atop TL II [DSS06]. All of the experiments are performed using the lazy acquire mode in TL II.

- The *Labyrinth* benchmark is a maze routing application. It uses Lee's algorithm [Lee61] to find the shortest-distance routes for a set of sources and destinations in a three dimensional matrix. The algorithm expands from the source point using a breadth first search. The search is guaranteed to reach the destination and a reverse back search formulates the route. The maze size used in the benchmark is 512*512*7. The routing process involves many memory writes that can have an enormous impact on performance if all are performed in the shared memory. In order to improve performance, the program is optimized by first copying the matrix to a local array through non-transactional reads and doing all subsequent computation on the local array. When a route is found, it is written back to the shared memory. The version provided within STAMP benchmark suite is the uncomposable version. We created two composable versions by changing the local array copy to fast reads. The pure TM version is created by changing the local array copy to transactional reads.

- The *Genome* benchmark is a gene sequencing program. It takes a number of DNA segments and matches them to reconstruct the original genome. In the phase one of the benchmark, all segments are put into a hash set to remove segment duplicates. To allow concurrent access, the hash set is implemented as

a set of unique buckets, each of which is implemented as a linked list. In our experiments, we created different versions of the Genome by modifying the list data structure.

- The *Intruder* benchmark is a signature-based network intrusion detection application. It scans network packets for matches against a known set of intrusion signatures. The main data structure in the capture phase is a simple non-transactional FIFO queue. Its reassembly phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session. We modified the list data structure used in the dictionary to create the different versions of Intruder.

- The *Vote* benchmark simulates a voting process. It supports three operations — vote, count, and modify. The underlying data structure maintaining the voting information is a binary search tree. Each node contains a two fields — voter and candidate voted by the voter. The `vote(ssn, candidate)` transaction casts a vote for a candidate on behalf of the voter with his ssn. The `count(candidate)` transaction returns the total number of votes a candidate has got. This transaction is a nested transaction composed of two transactions - verify and cast_vote. The `verify(ssn)` transaction verifies if the voter with a given ssn has voted. The `cast_vote(ssn, candidate)` transaction casts the actual vote if this voter has not voted yet. The `modify(ssn, candidate)` transaction changes a voter's vote to the new candidate if the voter with a given ssn exists. Since `vote(ssn, candidate)` is a nested transaction, the uncomposable version does not work correctly here. So we created three different versions of the benchmark: pure TM (where shared data is accessed through transactional memory interface),

and two versions that access the data through our fast reads composable interface (with both the lookup scheme and the partial commit scheme as the underlying implementation). There are 65,536 possible unique voters. The mix of operations of count, vote and modify is 10%, 80% and 10%.

- The *set* benchmark is an application that implements a set using a sorted linked list. It supports three operations and each is implemented as a transaction - insert, remove and lookup. `Insert(key)` inserts a key to the set. `Remove(key)` removes a key from the set. `Lookup(key)` searches for the key in the set. In our experiments, there are all together 512 possible unique keys in the set. The operation mix of insert, remove and lookup is 10%, 10% and 80%.

- The *nested set* benchmark is a nested version of application above. It has three nested transactions, nested insert, nested remove, and nested lookup. Each of the nested transaction has two of the corresponding single transactions within it. For example, a nested insert transaction is shown in Figure 6.4. The number of unique keys and operation mix is the same as the *set* above.

6.5 Discussion

In Labyrinth, Set, Vote, and Genome we observe that the optimized version's performance improves by a significant margin relative to a pure TM implementations. We also observe some performance overhead compared to the direct read version in cases where an uncomposable version exists.

Labyrinth is the application that shows the largest performance gain, clearly illustrating that certain types of applications can achieve enormous performance benefits by applying application-specific optimizations. In fact, the pure TM version of the

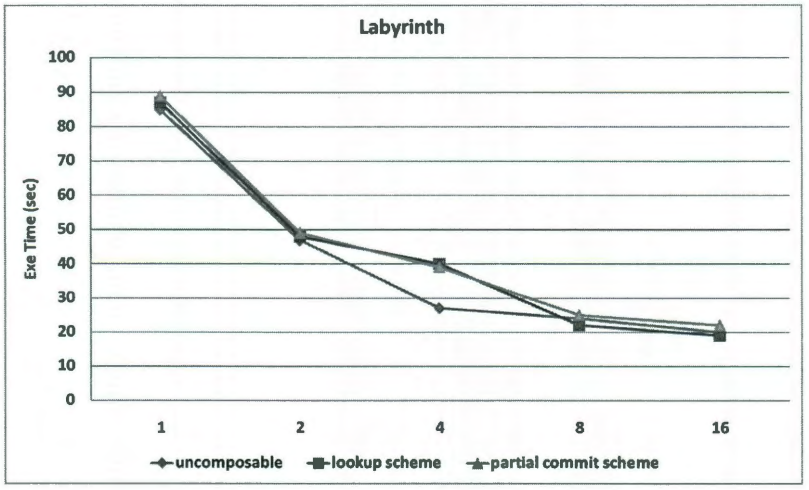


Figure 6.11 : Labyrinth execution time results. The figure shows the execution times of Labyrinth with different thread numbers using three different schemes.

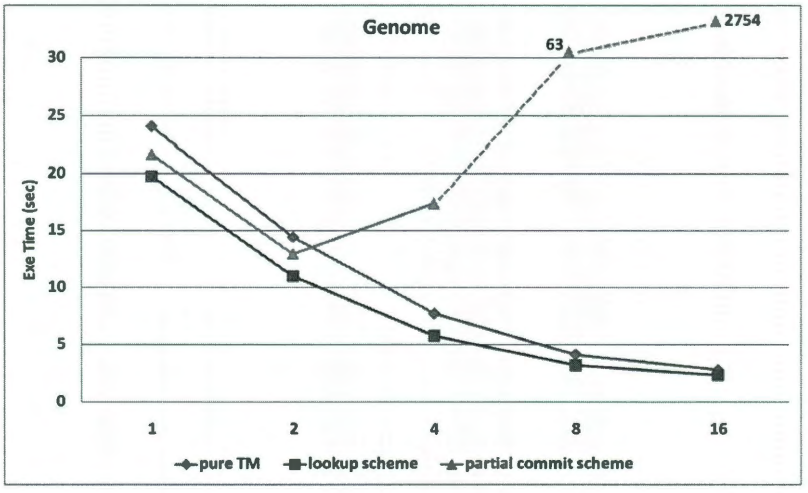


Figure 6.12 : Genome execution time results. The figure shows the execution times of Labyrinth with different thread numbers using three different schemes.

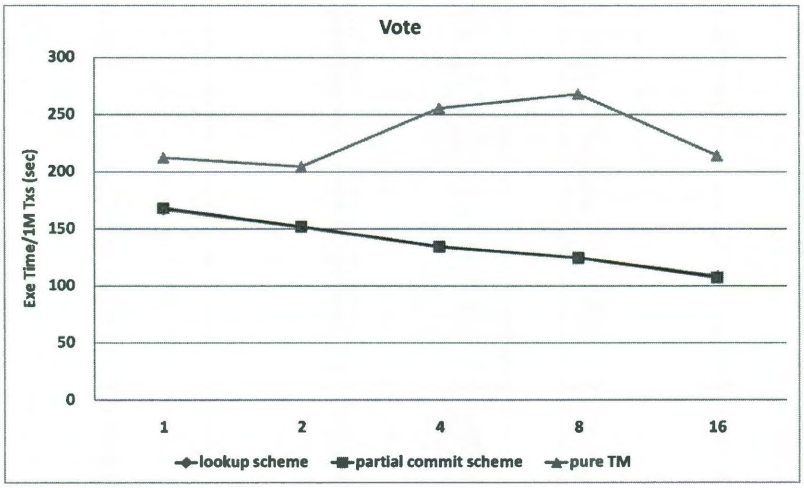


Figure 6.13 : Vote execution time results. The figure shows the execution times of one million transactions of Vote with different thread numbers using three different schemes.

Labyrinth version runs so slowly that we were unable to finish the experiments for cases with more than 4 threads. The two thread case takes over a day to complete (compared to about 50 seconds for the hand-optimized uncomposable version and the composable version using our TM interface extension). This is because many of the transactional reads in the pure TM version cause an immense number of conflicts and effectively create a live lock in the application. For this reason, we have omitted the pure TM results for Labyrinth in Figure 6.11, in order to illustrate the performance differences between the uncomposable optimized version of the benchmark and the composable version using our TM interface.

In Genome, the list operations are either a stand alone transaction or a part of a nested transaction that can be easily shown to be independent of other list operations. As shown in Figure 6.12, we observe a performance improvement of 26%

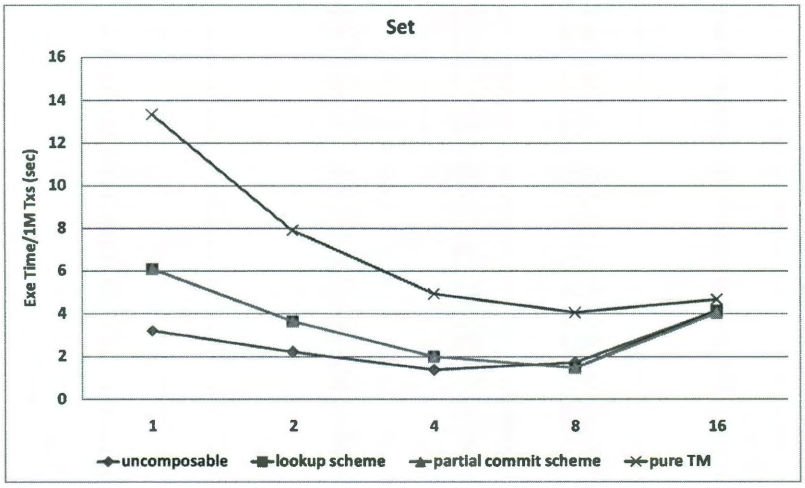


Figure 6.14 : Set execution time results. The figure shows the execution times of one million transactions of Set with different thread numbers using three different schemes.

in the 4 thread case using our lookup scheme. Across all thread counts, we observe a performance improvement ranging from 18% to 26%. The lookup scheme fares better in this benchmark than the partial commit scheme. The partial commit scheme encounters performance issues for more than 4 threads and the results are clipped in Figure 6.12.

Vote has a nested transaction `vote` that is composed of a `lookup` transaction to verify whether the voter has voted and an `insert` transaction that casts the actual vote. Figure 6.13 shows the results. We observe a performance improvement up to 150% when comparing our composable version with the pure TM version. There is no uncomposable version of this benchmark. The results of lookup scheme and PCM scheme are very close and are almost overlapped on the graph.

For the set benchmark based on a sorted linked list, Figures 6.14 and 6.15 show

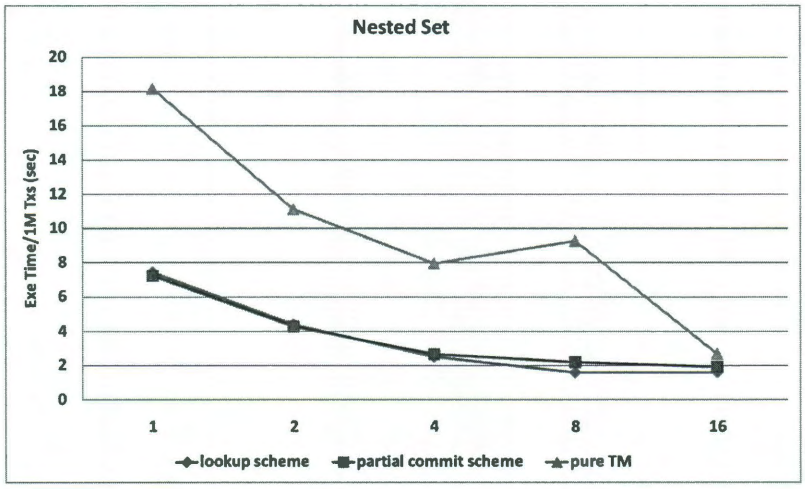


Figure 6.15 : Nested set execution time results. The figure shows the execution times of one million transactions of nested set with different thread numbers using three different schemes.

the results for single and nested transactions. We observe up to a 1.8x performance improvement over the pure TM version with our lookup scheme. We also observe that the performance overhead compared with the uncomposable version is up to 47% for the single thread case. The overhead is majorly from the extra work of maintaining the fast read set and merging it to the transactional read set. The performance overhead decreases as the amount of parallelism increases which indicates better scalability. For the case of nested transactions, the performance improvement is up to 1.3x. Though the two transactions nested clearly have no dependencies, the benchmark is implemented by assuming they might and inserting $TxFlush$ between them. Better performance could be achieved if the programmer were to take advantage that there is no dependence between transactions and remove the $TxFlush$ call.

The last benchmark, Intruder, does not show performance improvement (Fig-

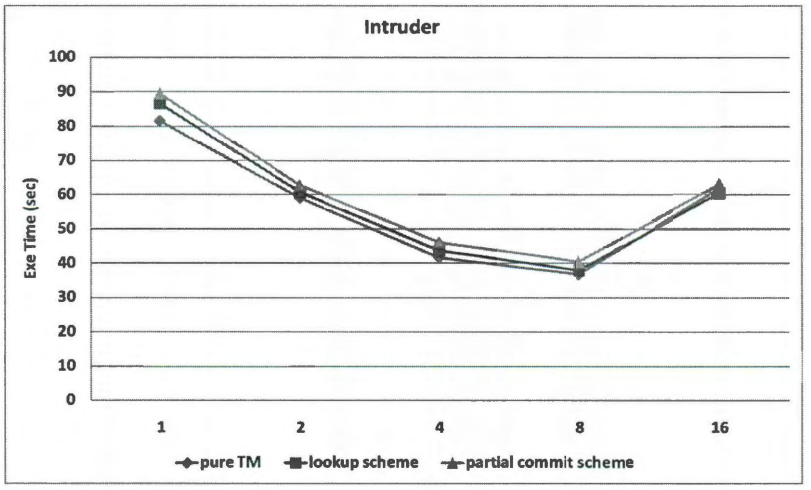


Figure 6.16 : Intruder execution time results. The figure shows the execution times of Intruder with different thread numbers using three different schemes.

ure 6.16), but rather a minimal (couple of percent) performance degradation when compared to the pure TM version. The reason is that the list operations in Intruder are only used in nested transactions and infrequently at that. The trade-off of spending additional time to set up a separate read set and merge it to the transactional read set does not pay off in this case. Note that an uncomposable version of this benchmark does not even exist.

Following are some of the design choices and issues that arose while developing the TM interface extension, in no particular order:

- *Lookup scheme vs. partial commit scheme*: In our experiments, the lookup scheme outperforms the partial commit scheme in most cases. The larger time window in which we hold locks in the partial commit scheme increases the possibility of a conflict. The advantage of the partial commit scheme is that

it does not require later nested transactions to search the past write set. For applications that feature an expensive search process, we expect that the partial commit scheme would achieve better performance.

- *Non-in-place update STM*: We only evaluate an in-place update TM system in our experiments. For STM systems that do not use in-place update, similar problems will arise when the programmer wants to read directly the visible copy. We will explore non-in-place update STM systems in the future.

- *Fast Read Advantages and Disadvantages*: Our fast read extension addresses the problem that application-specific optimized transactions do not integrate with the STM runtime/library to provide composability guarantees. Under the hood, the extension provides the necessary link between the programmer and the consistency guarantee mechanism integrated in the STM system. Therefore the places to use the extension are similar to ones where the original optimizations apply. The major performance incentive of the optimization is to eliminate part of the time-consuming validation work involved in a pure STM implementation. If an application's consistency depends on most of its past reads, the work it can save could become rather limited and might not be worth the effort of developing an optimized version.

- *Pure TM version implementation of Labyrinth*: The pure TM version of Labyrinth is implemented by first copying the entire matrix to a local version transactionally. Then, routing is performed in the local matrix. The route found is written back to the global matrix. This is very similar to the original algorithm used in the Labyrinth benchmark.

  A second approach would be to perform the routing computation in the global

matrix directly. The first step of the routing algorithm is a breadth first search from the source to the destination. In this phase, the each cell visited is marked with a distance to the source. The first step stops when the destination is found. The second step of the algorithm searches from the destination backward to the source for the shortest path from source to destination. The third step is to clean up the marks left by the first step. We expect that this approach would perform just as bad as the first one, since the updates of marks in the first and third step will cause an enormous number of conflicts and the application would be very vulnerable to live locks. We implemented the first approach in our experiments.

- *TxFlush Programmer Visibility*: For correctness, a $TxFlush$ is required between every transactional write to a memory location *loc* and a subsequent fast read to *loc*. If $TxFlush$ is exposed to the programmer, he/she can reason about the dependencies between nested transactions and only insert $TxFlush$ where strictly necessary. Alternatively, one can envision a compiler that inserts $TxFlush$ before and after every nested transaction that uses the fast read interface, then analyzes the dependencies between shared memory accesses and removes the unnecessary $TxFlush$ calls, further improving the performance of our proposed techniques. In this chapter, we have used a straightforward and conservative approach that inserts $TxFlush$ before and after every nested transaction that uses the fast read interface.

Chapter 7

Distributed STM Time Base

In this chapter, I present a time base design to solve the potential scalability bottleneck of the time base designs in current STM systems.

7.1 Introduction

Time-based STM systems use timestamp information to reason about the ordering among operations on shared resources and provides sufficient proof that a transaction is consistent. Timestamping can be used to provide sufficient support about transaction consistency like in TL II. It can also be used to work with another consistency guarantee technique such as validating past reads to increase the accuracy of the validation process. In recent systems, the addition of the timestamping technique effectively reduces early STM system performance overhead that used incremental validation. As a fast and efficient technique of providing consistency guarantee, this technique is widely used in recent STM systems such as TL II [DSS06] and LSS [RFF06].

A critical component of time-based STM systems is the time base that generates all timestamps in the system. Every timestamp operation consults the time base for the current time or a new timestamp and this happens very often during the execution of a TM application. The way the time base is designed is closely related to the performance of the STM system. The time base in existing STM systems use a

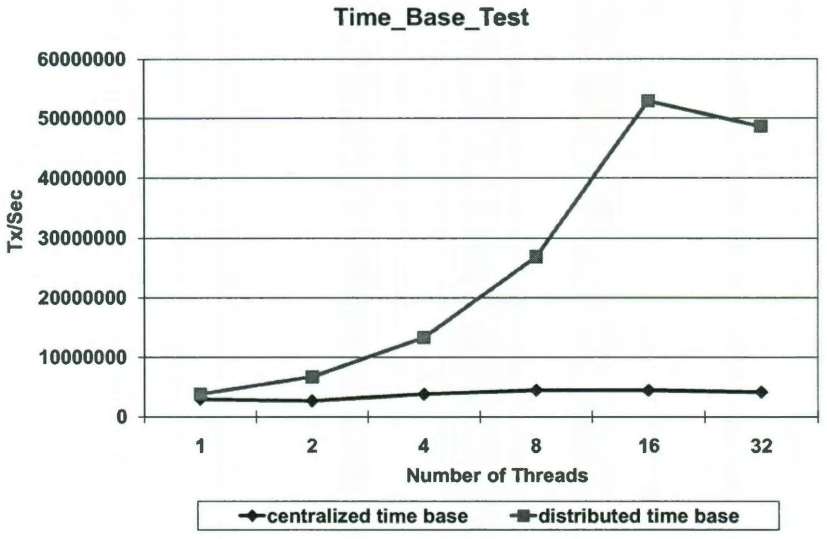


Figure 7.1 : Throughput comparison of the Time_Base_Test benchmark using a distributed time base and a centralized time base. The figure shows that the distributed time base outperforms the centralized time base.

```
// thread_ID: the ID of the owner thread.
// A: an array containing integers for each thread.
// Integers in A are padded to avoid false sharing.
// stop: a variable that controls when to terminate.

time_base_test(thread_ID, stop, A) {
    while (stop) {
        atomic {
            A[thread_id]++;
        }
    }
}
```

Figure 7.2 : Time_Base_Test implementation.

single integer as the time base shared by all transactions. But the centralized nature of a single global time base makes the time base a potential scalability bottleneck.

Figure 7.1 illustrates the potential scalability bottleneck problem using a synthetic application Time_Base_Test shown in Figure 7.2 developed by me. In Time_Base_Test each thread updates a separate memory location so transactions do not conflict. All transactions are update transactions and very short. The results shown in Figure 7.1 are collected on a Niagara 1 computer. The two lines on the figure represent the throughputs of Time_Base_Test at different thread numbers based on two different time base designs – the centralized time base and the distributed time base that is discussed later in this chapter. The figure clearly shows that the throughput using the centralized time base does not scale, but using the distributed time base gives a much better scalability. Some techniques have been proposed to reduce the contention on the centralized time base. For example, Dice, Shalev and Shavit proposed to associate the thread ID with each timestamp so the update to the time base can be performed with one CAS [DSS06]. We also show several optimized commit phase designs in chapter 5. But the existing techniques all use a centralized time base. In this chapter, we propose a distributed timestamp design that exhibits better scalability for certain benchmarks. We evaluated our design using STAMP benchmarks on a Niagra II computer with 64 hardware threads. The results show the performance of our distributed time base design is very competitive with the centralized time base design and can outperform the centralized time base design for some applications. Our distributed time base design is superior for highly concurrent applications.

Logical clocks have been extensively researched in the distributed computing community. The lack of centralized resource in distributed systems and the difficulty of having accurate synchronized physical clocks make logical clocks an appealing technique to coordinate distributed events and states. By carefully designing the way to map events in a distributed system to their logical clock states, the states of logical

clocks can be used to provide sufficient information to determine all or some of the causal relationships of the distributed events. Various methods of designing logical clocks have been proposed. In 1978, Lamport [Lam78] first proposed the concept of *logical clocks*. Lamport clocks capture a partial ordering between events in distributed systems. They consist of a mapping from events to the set of integers that in principle captures the causal order between events. But Lamport clocks do not detect the concurrency between events. Later, Fidge and Mattern [Mat89] proposed the use of vector clocks to accurately capture the causal relations between distributed events. In vector clocks, each distributed entity has its own vector slot and each message contains the vector clock of its host. Vector clocks have better accuracy but require a vector that is O(n) where n is the total number of hosts in the system. Torres-Rojas and Ahamad [TRA96] proposed a class of clocks that do not characterize causality completely, but are scalable because they can be implemented using constant size structures. The scalability that plausible clocks focus on is the space scalability. Though the aforementioned clocks in the distributed computing like here time-based STM systems use the same principles. This chapter focuses on the scalability problem of the time base used in STM systems. Though our proposed solution shares similarities with the timestamps used in distributed systems, it is used to solve a problem different from the space scalability problem. Also, STM systems are designed to work in a more centralized environment such as multi-core processors, where efficient shared resources are readily available. How the time base should be designed for STM is a topic that no previous work has explored.

In this chapter, the time base refers to the data structure that maintains and generates logical timestamps in STM systems. Existing time base designs for STM all use a shared integer counter. Due to the lack of centralized resource in distributed

systems, logical clock designs originated from the distributed computing community usually do not use centralized resources to generate the timestamps. Instead, the responsibility of creating and maintaining the time base are distributed across different sites in the distributed system. The nature of logical clock designs can be different for transactional memory because transactional memory is designed with a focus on multi-core processors, where centralized shared hardware resources such as cache and memory are readily available.

In section 7.2 we discuss several time base designs including the distributed time base design. Then in section 7.3 we present and discuss experimental results.

7.2 Time Base Designs

In this section, we focus on the design of a time base. Clocks used in these designs can all be categorized as some form of logical clocks. However, he time base designs have different characteristics to consider for STM due to the availability of efficiently available shared resources. We describe six different time base designs. They are a centralized single counter time base, a centralized vector time base, a centralized tree time base, a distributed time base, a distributed time base with synchronization, and a centralized timestamp vector time base with synchronization.

- *1. Centralized single integer counter time base:* The time base of this design is a single shared integer counter. This is the time base design that most existing time based STM systems use, such as TL II and LSS. In this design, all operations of the time base are directed to the same shared counter. Concurrency control is necessary to coordinate the operations on the shared counter for correctness. All reads and writes of the time base contend with each other.

Time base definition: int counter;

- *2. Centralized vector time base:* The motivation of this design is to reduce the contention over the single shared counter time base. Similar to the first design, the time base of this design is also centralized. The difference is that the time base of this design is a vector instead of a single counter. Each thread in the system has its own slot in the vector and all updates are performed in its own slot. Therefore the updates are not competing with each other and can be performed without concurrency control. To get a timestamp of the time base, a transaction sums the entire vector by scanning through it. To avoid false sharing, each slot in the vector should be located in its own cache line.

  The advantage of this design is writes to the time base are not contended. But this comes at the expense of the time base reads since they need to scan the entire vector. Comparing with the single read in the single counter design, reading the time base in this design is more expensive and is linear in the number of threads. This makes this scheme more suitable for cases where the number of writes overwhelms the number of reads. But in a typical transaction execution, a transaction first reads the time base. It might perform extra reads when validating past reads. And a transaction only needs to perform at most one update to the time base when the transaction is an update transaction. So typically the number of reads of the time base is more than the number of writes. This makes the performance tradeoff of this design by penalizing reads in favor of writes less attractive. So in this chapter, we will not experimentally evaluate this design.

  Time base definition: cache_line_size_int counter[n]; where n is the number of

```
typedef struct tmp_counter {
    int counter;
    int padding[(cache_line_size-sizeof(int))/sizeof(int)];
} cache_line_size_int;
```

Figure 7.3 : Definition of a cache line size int.

threads in the system.

In the above definition, cache_line_size_int is a data type whose size is the same as the cache line size. Figure 7.3 shows a definition of cache_line_size_int in C/C++.

- *3. Centralized tree structured time base:* The goal of this design is to reduce the overhead of reading the time base in the second design. In the second design, a transaction reads the entire vector that includes p elements for every read, where p is the number of threads. In this design, a transaction only needs to read a fraction of p slots for a time base read. The idea is to let the writes take more responsibility for summing up vector slots so reads of the time base can perform less work.

  In this design, the time base is structured as a tree with the leaf nodes being the places each thread updates. Similar to the second design, each thread has its own leaf node and all updates from the same thread are performed in the same leaf node. Therefore at the leaf level, updates to the time base are not contended. A parent node in the tree contains the sum of all its children nodes. Each time the base update not only updates its leaf node but also its parent node. The parent node contains the sum of all its children. The number of

ancestors one single leaf update need to propagate to is tunable. The updates to ancestors are contended but due to the smaller number of children to update the node, the contention is smaller than a single counter. A read of the time base now only needs to read the nodes in the frontier to which updates propagate.

The degree of the tree can be any number from 2 to the number of threads p and does not have to be the same across the tree. The number of leaves in a subtree below a frontier and the branch factor of the tree determine the level of contention a parent node has to take.

Compared with the second design, this design penalizes the write operation in favor of the read operations. It can decrease the contention on updating the time base, but its complex data structure and complicated procedure to update the time base makes its performance not attractive on an STM system for multi-core systems. In this chapter, we will not evaluate its performance either.

Time base definition: Tree counter;

- *4. Distributed time base:* This design discards all centralized shared data structures in the time base. All data related to the time base is distributed across all participating threads. The distributed design reduces the amount of cache coherence traffic from operations on a centralized time base. Also this design makes better use of spatial locality and does not use expensive atomic operations such as CAS when manipulating a local timestamp vector. For a centralized time base like designs 2 and 3, shared data structures are usually modified using atomic operations such as CAS and are designed to avoid false sharing and no spatial locality is utilized.

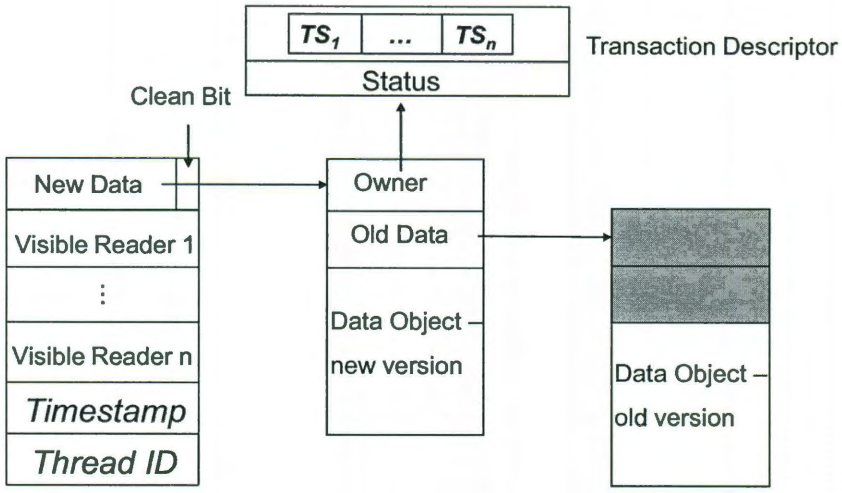


Figure 7.4 : Distributed time base metadata.

In this design, there is no shared time base. Figure 7.4 shows the distributed time base metadata based on RSTM2 [MSH+06]. Each thread maintains a local vector of timestamps. The timestamp vector records the latest timestamps it observes from its thread's memory accesses. Each shared object is associated with a tuple of timestamp and thread ID. Every time a transaction of a thread K updates a shared memory location, the transaction increments the local timestamp of thread K by 1 and puts this new timestamp and the thread ID K into the object. When a transaction reads a shared memory location, it compares the embedded timestamp with the corresponding timestamp within its timestamp vector. The local timestamp vector is only accessed locally and can be optimized for spatial locality.

The disadvantage of this design is it potentially reports more false positive

transaction inconsistent states. This is due to two reasons. First, a timestamp read by a transaction does not precisely represent the current timestamp of the thread that wrote the timestamp. This is because writes performed by the same thread put the up-to-date timestamp at the moment of the write. Second, when a transaction reads a timestamp from a shared location, the timestamp only represents the time information of the thread that last modified this location but not any other threads. Compared with the centralized time base design, all information of the past writes are kept in the same shared time base and read by all other transactions.

Also in this design, an update to a shared object overwrites its previous timestamp. This causes the previous time information to be lost for future reads. There are ways to reduce the information loss such as using vector timestamps but at a cost of both increased space overhead and maintenance overhead.

The distributed design is suitable for applications that are highly parallel where a shared time base becomes a scalability bottleneck.

- *5. Distributed time base with inter-thread synchronization:* The motivation of this design is to reduce the number of reported false positive transaction inconsistent states in design 4 by synchronizing local timestamp vectors to make them closer to the global time. The idea is to introduce inter-thread time synchronization. In this design, a thread periodically chooses a set of threads to synchronize. The synchronization updates the local timestamp vector to the one that is more recent between itself and the synchronized one. But this design endures extra synchronization overhead that is expensive in a multi-core environment.

Various schemes can be designed to synchronize the timestamps. This problem is very similar to routing schemes in networking. Designs similar to unicast, multicast, broadcast, etc. can be used to synchronize the timestamp vectors. For example, all threads can be formed as a directed ring and each thread communicates with its predecessor. The threads can be organized into different structures and have different synchronization performance characteristics.

There are also various locations within a transaction to perform a synchronization. For example, one approach is to perform a synchronization with the thread that made the update when a thread encounters a new timestamp. This way can ensure the synchronization will get at least one new timestamp.

- *6. Centralized timestamp vector with inter-thread synchronization:* The motivation of this scheme is to increase the effectiveness of synchronization in design 5. In design 5, one synchronization can only get timestamps from a single thread. The synchronization might read old timestamps. This design reduces the number of old timestamps. It uses a centralized timestamp vector similar to design 2. Every thread has its own slot in the vector and only updates its own slot. A synchronization is now performed by reading the corresponding slots in the shared vector and updating each slot to the latest one. Compared with design 5, in this design a synchronization has better chance to get more up-to-date timestamps because of reading from the shared vector.

  The disadvantage of this design is the central timestamp vector needs to consider false sharing. The typical way of padding the vector slot to cache line size makes the synchronization step lose performance benefits from spatial locality.

  The time base definition of this design is the same as design 2.

The above six designs are several designs of the time base. Other designs are also possible. Each of these designs has different performance characteristics. When and where to use which design is closely related to system parameters and application characteristics. From preliminary experiments, we found that design 2, 3, 5, and 6 do not show competitive performance in STM systems compared with the centralized single counter time base design due to their high overhead of operating the time base. In this chapter, we only evaluate design 4 on a broader range of benchmarks from STAMP to compare its performance with the centralized single counter time base design.

7.3 Experimental Results

In this section, we evaluate the performance of the distributed time base design and compare it with the centralized single counter design.

The algorithm of reading a shared object for the distributed time base design is shown in Figure 7.5. The open function checks if the timestamp within the object is younger than the timestamp kept in its local timestamp vector. If the timestamp is younger, a validation of the read set is performed together with updating the local timestamp vector.

We implemented the distributed time base design and a lazy snapshot version based on transactional locking II. We compared the performance of the distributed time base design with the original TL II and the LSS version. The machine for the experiments has one single Niagara 2 processor. The processor has 8 cores and each core has 8 hardware threads.

The benchmarks we used in the experiments are from STAMP. They include Bayes, Labyrinth, SSCA2, etc. For each benchmark we collected results for 1 to 64

```
// T: transaction descriptor.
// O: object being opened.
// m: mode - read only or write.
// T.O: read set of transaction T.

open(T, O, m) {
    if (m == write) {
        T.update = true;
    }
    if (O.ts > T.ts[O.id]) {
        T.ts[O.id] = O.ts;
        validate(T.O);
    }
    T.O = T.O union O;
}
```

Figure 7.5 : The open function when using the distributed time base design.

threads at a pace that doubles the number of threads every time. The results reported here are the average of three separate runs.

In the experiments, we compare the centralized single counter time base design and the distributed time base design. We compare the centralized single counter results from TL II and lazy snapshot with the distributed time base design.

7.4 Discussion

The distributed time base is designed to reduce the contention over updating the shared time base. When the thread number is low, the contention over the shared time base is not high, thus the effect is not expected to be significant. Also the contention over the shared time base is closely related to the application's characteristics. More threads do not imply higher contention for the time base. For example, if the

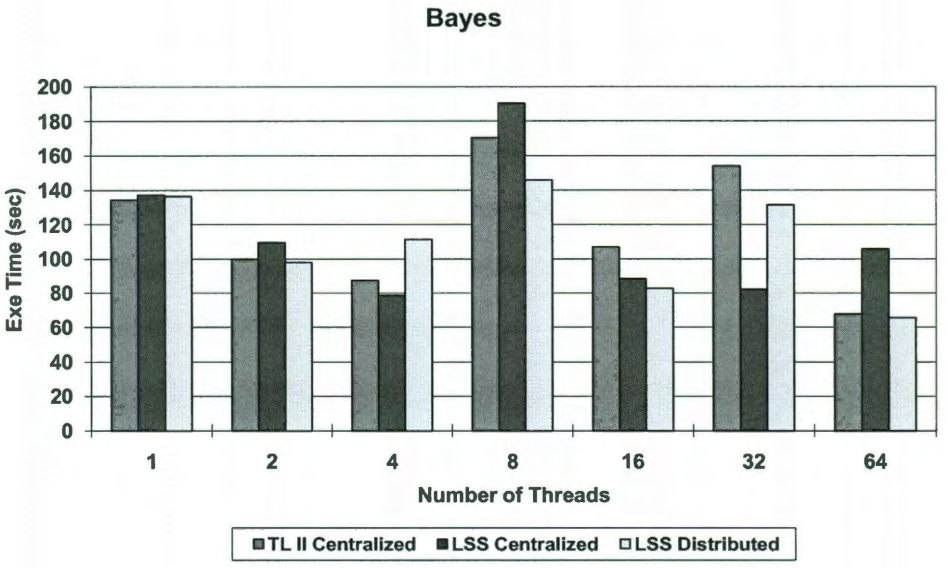


Figure 7.6 : Bayes performance results.

application is highly sequential such as updating a shared resource, even when the application uses many threads, the pressure on the time base is still low because the bottleneck is not on updating the time base but on the application itself. For applications whose scalability bottleneck is not the shared time base, using the distributed time base will not help. On the other hand, using the distributed time base may slow down the application because the distributed time base provides less accurate information than the centralized single counter.

Based on the experimental results, the distributed scheme does not show a big performance boost across the board, but it shows very competitive performance in several benchmarks.

Bayes shows a lot of variation in its results as shown in Figure 7.6. Bayes uses an alternating decision tree [FM99] under the hood. It has long transactions with large read and write sets. Its transactions have a high probability of conflicting. The

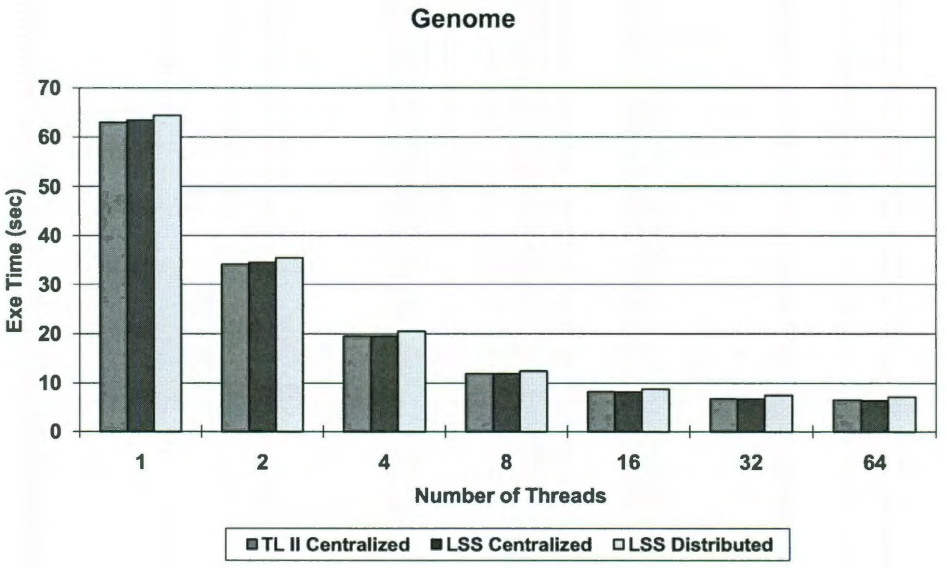


Figure 7.7 : Genome performance results.

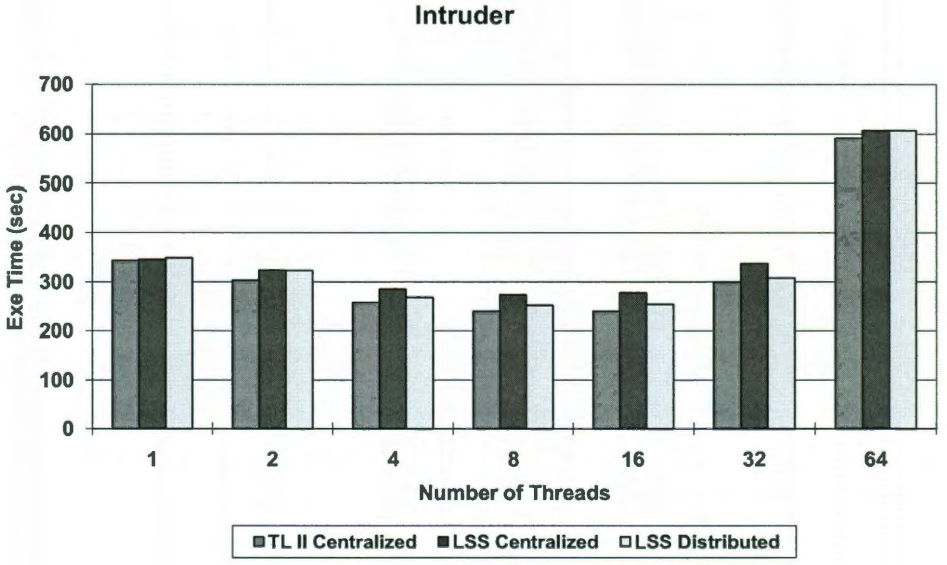


Figure 7.8 : Intruder performance results.

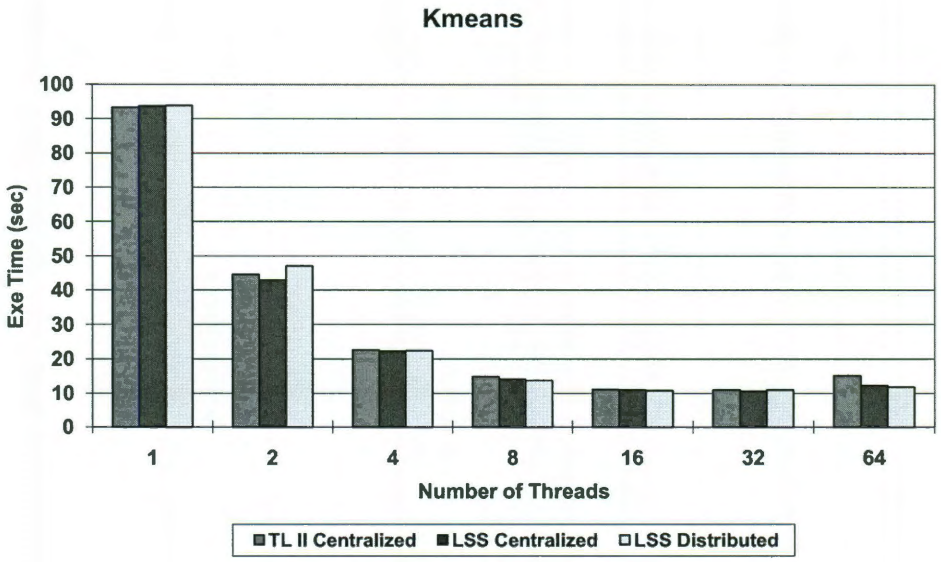


Figure 7.9 : Kmeans performance results.

distributed time base on average performs close to the average of centralized time base. For the thread case 2, 8, 16, 64, LSS distributed beats both centralized TL II and centralized LSS. For 4 and 32 threads, the distributed time base version loses by 3%.

In Genome, distributed time base version shows a minor performance degradation for 1 to 64 threads of about 2-3% as shown in Figure 7.7. Transactions in Genome have moderate read and write set sizes and show little contention.

In Intruder, the main data structures used are a FIFO queue and a balanced tree. The benchmark spends moderate amount of time in transactions. The distributed time base version shows a modest performance improvement compared with the LSS centralized in most thread numbers with up to 8.7% improvement as shown in Figure 7.8. But compared with the TL II version with centralized time base, the distributed version shows minor performance degradation in all thread numbers.

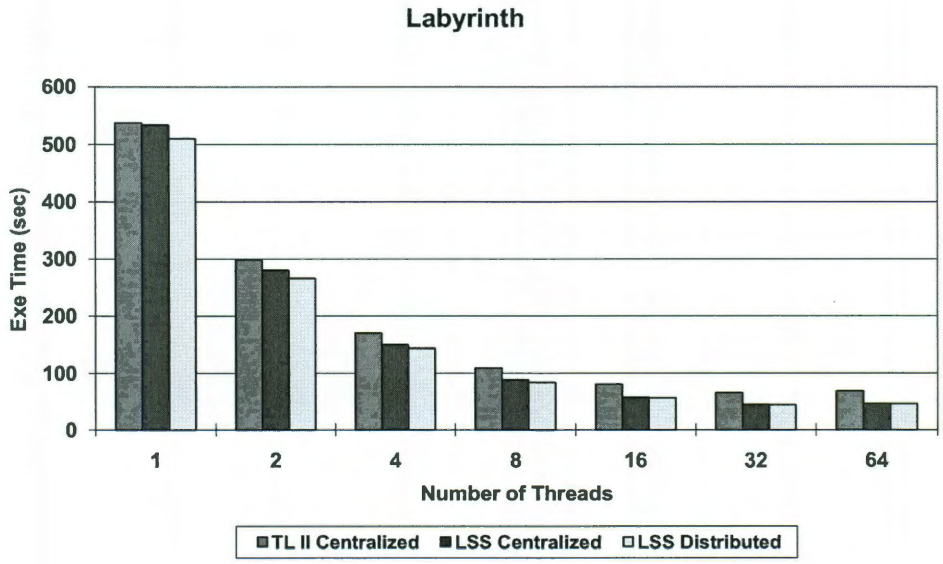


Figure 7.10 : Labyrinth performance results.

Kmeans implements a clustering algorithm for objects in an N-dimentional space. Kmeans does not spend a lot of time in transactions. The read and write sets are small. The results for Kmeans are mixed. The distributed version shows performance improvement for bigger thread numbers of up to 4% but shows performance degradation for 2 and 4 threads as shown in Figure 7.9.

The Labyrinth benchmark implements the Lee algorithm [Lee61]. The benchmark is optimized using user level direct reads discussed in chapter 6. The privatization technique used in Labyrinth greatly reduces the amount of conflicts and therefore improves the performance. Our distributed version shows performance improvement up to 5% compared with the original LSS version as shown in Figure 7.10. Compared with the original TL II version, our distributed version shows up to 32% performance improvement. The improvement compared with the original TL II version is the result of using our distributed time base design and the LSS algorithm in our distributed

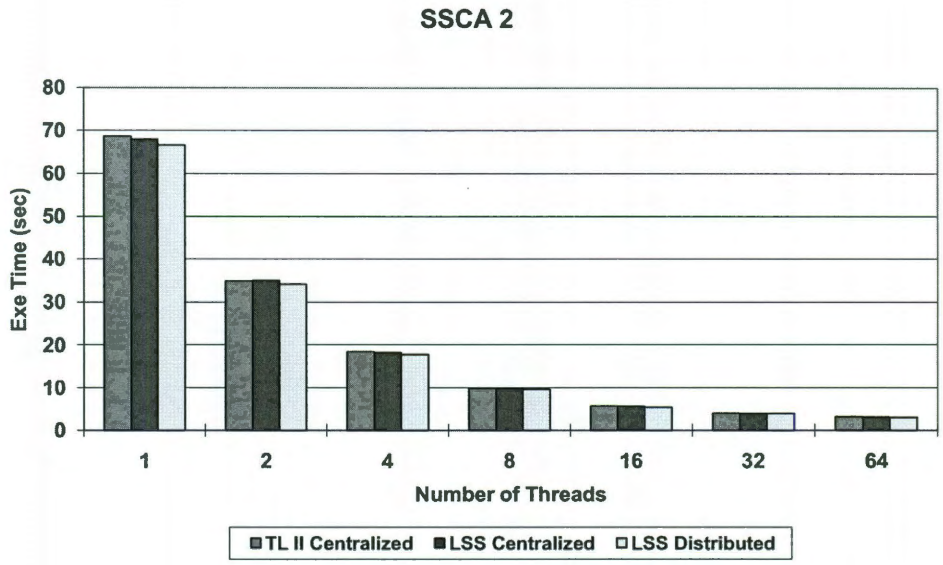


Figure 7.11 : SSCA2 performance results.

time base version. We contribute the improvement majorly to the use of LSS here.

SSCA2 is the first kernel in the scalable synthetic compact applications operating on a large, directed and weighted multi-graph. The first kernel uses adjacency arrays and auxiliary arrays to build an efficient graph data structure. Transactions are used to add nodes to the graph in parallel. The transactions in SSCA2 are short. The read and write sets of each transaction is also small. The contention is also low. In SSCA2, the distributed time base shows performance improvement of up to 3.3% for 16 threads as shown in Figure 7.11.

Vacation shows very similar performance for all three schemes. The performance differences are within a couple of percents as shown in Figure 7.12.

In summary, the results of the distributed time base on STAMP benchmarks are mixed. The distributed time design is able to show performance gains in SSCA2 and Labyrinth compared with the centralized counterparts. For other benchmarks, the

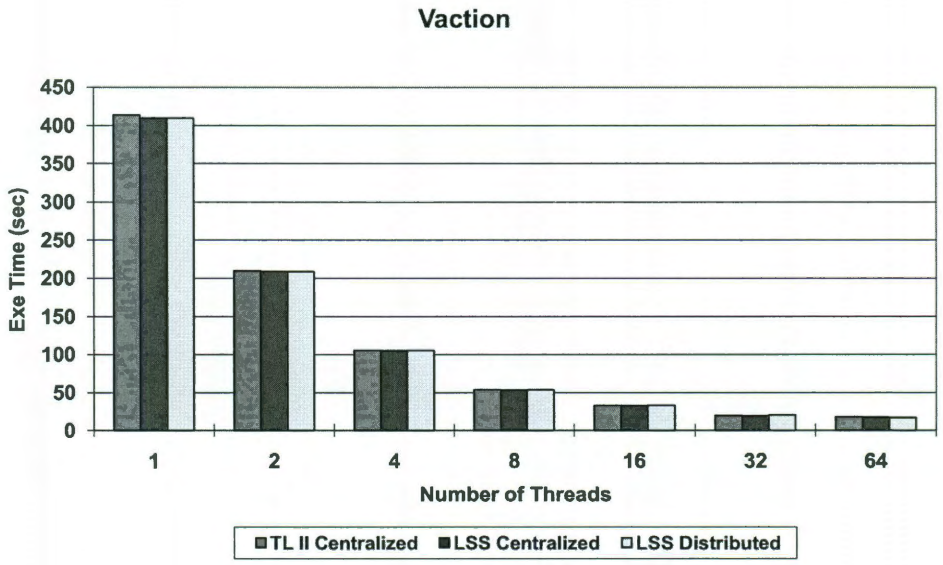


Figure 7.12 : Vacation performance results.

distributed time base show limited performance loss to one of the two counterparts. But as we analyzed earlier, the design is to solve the potential scalability bottleneck of the centralized time base. And the availability of such types of benchmarks in STAMP is an evidence for its usefulness and we expect other applications can benefit from this design.

Chapter 8

Multi-place Isolation

8.1 Introduction

The computer industry is entering a new era of mainstream parallel processing due to current hardware trends and power efficiency limits. Now that all computers— embedded, mainstream, and high-end — are being built using multi-core chips, the need for improved productivity in parallel programming has taken on a new urgency. One of the major obstacles to improved productivity is the complexity of coordination and synchronization of parallel tasks that is inherent in current parallel programming models combined with the complexity of distributing computation and data across future many-core processors. Coordination and synchronization constructs can take many forms in practice such as mutual exclusion in accesses to shared resources using locks, termination detection of child threads using join operations, collective synchronization using barriers, and point-to-point synchronization using semaphores. Recent efforts on productivity improvements in coordination and synchronization include transactional memory systems for mutual exclusion [HM93], work stealing schedulers to support dynamic nested parallel execution model with lightweight task creation and termination detection as in Cilk [BJK+95], and deadlock-free synchronization for dynamic parallelism as in X10's clocks [CGS+05] and Habanero phasers [SPSS08]. These approaches have motivated new research on delivering efficient parallel performance, while retaining safety guarantees that are important for productivity. In

contrast, there has been relatively less attention paid to the additional challenges that arise from distributed data and computation *e.g.,* most transactional memory systems assume that all shared data is uniformly accessible by all threads. A notable exception has been the three languages initiated in the DARPA High Productivity Computing Systems program — Chapel [Cra10], Fortress [ACH+08], and X10 [CGS+05] — all of which are exploring approaches for more productive forms of coordination and synchronization in conjunction with capabilities for distributing data and computations. Though the initial focus of these languages was to target high-end cluster-based supercomputers, the distribution constructs will also be relevant to future many-core processors where locality and data placement will be important challenges.

In this chapter, we focus on the problem of delivering productivity and scalable performance for mutual exclusion coordination among dynamically scheduled tasks. It is widely accepted in the community that the use of an `atomic` statement is more productive for the user than explicit lock management. As an example, all three HPCS languages include `atomic` statements. However, there are many different opinions on what mechanisms should be used to implement an `atomic` statement. There has been a lot of attention focused recently on mechanisms based on software and hardware transactional memory, but this direction has also led to a number of performance challenges as well as semantic challenges when the `atomic` construct is augmented with explicit transactional commands such as `abort` and `retry`. It is also possible to use system-generated locks as an alternate mechanism to implement atomic sections, either using a simple coarse-grained approach with a single lock or by using compiler analysis to use multiple locks [ZSZ+08]. Lock-based approaches often perform better than transactional approaches in the presence of low contention, but do not scale as well when the possibility of contention increases.

Our solution to this problem is to extend the *place* construct in X10, which supports distribution and alignment of tasks and data objects. The `atomic` statement in X10 guarantees *single-place atomicity* because a task (activity) is only permitted to access place-local data within the `atomic` statement. Though it provides a sound semantics, it is restrictive for computations where an atomic statement may need to access data located in more than one place. In this chapter, we extend the X10 model with *multi-place isolated statements* of the form `isolated(`⟨*place-list*⟩`)` and `isolated(*)`, and generalize X10's locality rule to require that all data accessed within a multi-place isolated statement be local to any one of the places in the statement's scope. Thus, the scalability of `isolated` statements can be improved by increasing the number of places. The *two-level lock-based* implementation of multi-place isolation presented in this thesis is guaranteed to be deadlock-free and shows scalability improvement by increasing the number of places. For a simple Sorted Linked List example, the following improvements in throughput were observed relative to 1 place on three different hardware platforms — up to 2.1× on a dual quad-core Intel Xeon SMP with 8 cores, up to 4.9× on a Sun Niagara 2 SMP with 8 cores and 64 hardware threads, and up to 2.6× on a SunFire 6800 server with 16 UltraSPARC III processors. Note that a lock-based implementation of multi-place isolation is different from fine-grained locking because the programmer does not have to worry about synchronization and related errors in multi-place isolation and the approach offers the programmer a single tuning knob (number of places) to improve the performance of multi-place isolation.

The rest of the chapter is organized as follows. Section 8.2 re-caps the X10 approach to single-place atomicity. Section 8.3 describes our approach to multi-place isolation, and Section 8.4 outlines our current implementation approach based on

two-level locking. Section 8.5 summarizes our experimental results.

8.2 Single-place Atomicity in X10

In this section, we summarize the *place* and *atomic* constructs in the X10 language [CGS+05, Vij08] and discuss its *single-place atomicity* semantics.

8.2.1 Places

As defined in X10, a *place* [CGS+05, CSSB08], is a collection of non-migratory *activities* (lightweight tasks) and mutable *locations* (object fields and array elements). Places are virtual — the mapping of places to physical locations in a parallel system is performed by a *deployment* step that is separate from the X10 program. Though objects and activities do not migrate across places in an X10 program, an X10 deployment is free to migrate places across physical locations based on affinity and load balance considerations. While an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in remote places.

An X10 computation may have many concurrent *activities* in flight at any given time in multiple places. An asynchronous activity is created by a statement `async (P) S` where `P` is a place expression and `S` is a statement. The statement `async S` is treated as shorthand for `async(here) S`, where `here` is a constant that stands for the place at which the activity is executing (every X10 activity has a designated place).

X10 has a global address space. This means that it is possible for any activity to create an object in such a way that any other activity has the potential to access it. The address space is said to be *partitioned* in that each mutable location and each activity is associated with exactly one place, and places do not overlap. A *scalar object* in X10 is allocated completely at a single place. In contrast, the elements of

an *array* may be distributed across multiple places, and its *distribution* specifies the place for each element.

A mutable variable is said to be *local* for an activity if it is located in the same place as the activity; otherwise it is *remote*. Though a range of memory models is under consideration for X10 [SJMvP07], the latest definition of the language (X10 v1.7 [Vij08]) continues to support the *Locality Rule* introduced in X10 v0.41 [CGS+05] : an activity may only read/write local variables. Any attempt by an activity to read/write a remote mutable variable triggers a runtime exception (stemming in turn from a failed type cast). However, an activity may read/write remote variables asynchronously by spawning activities at their places.

X10 currently adopts a conservative design in which the number of places is fixed when an X10 program launches; thus there is no construct to create a new place. This is consistent with current programming models, such as MPI, UPC, and OpenMP, that require the number of processes or threads to be specified when an application is launched.

8.2.2 Atomic Blocks

An *unconditional atomic block* in X10 is a statement `atomic S`, where `S` is a statement. X10 permits the modifier `atomic` on method definitions as a shorthand for enclosing the body of the method in `atomic`. An atomic block is executed by an activity *as if* in a single step during which all other concurrent activities in the *same place* are suspended. An atomic block may include method calls, conditionals, and other forms of sequential control flow, but may not create a new activity or perform a blocking operation. Since the only way to access remote data is by creating asynchronous activities, the locality rule ensures that all accesses within an atomic block are to

local data.

An X10 statement may terminate *normally* or *abruptly*. A statement terminates abruptly when it throws an exception that is not handled within its scope; otherwise it terminates normally. For simplicity, the X10 `atomic` construct only guarantees isolation during normal execution. If `S` terminates abruptly, then `atomic S` terminates abruptly. Thus, atomicity is guaranteed for the set of instructions actually executed in the atomic block. If the atomic block terminates normally, this definition is likely to coincide with what the user intended. If the atomic block terminates abruptly by throwing an exception, then atomicity is only guaranteed for the instructions executed before the exception is thrown. If this is not what the user intended, then it is their responsibility to provide exception handlers with appropriate compensation code.

8.2.3 Single-place Atomicity

X10 supports a single-place atomicity model *i.e.,* the `atomic` construct is constrained to permit data accesses in a single place, specifically the place in which the activity executing the `atomic` block resides. Consider the following atomic block as a representative example of a linked list insert operation:

```
atomic {
    /* S1 */ newNode.setNext(currNode);
    /* S2 */ prevNode.setNext(newNode);
}
```

By declaring the block as atomic, the programmer maintains the integrity of a linked list data structure in a multithreaded context.

However, a `BadPlaceException` will be thrown if the `newNode` or `prevNode` object is non-local in statements `S1` and `S2` respectively. (In X10 v1.7, a `BadPlaceException`

```
public boolean insert(int v) {
    INode newNode = new INode();
    newNode.setValue(v);
    isolated (*) {
        INode prevNode = this.first;
        INode currNode = prevNode.getNext();
        while (currNode.getValue() < v) {
            prevNode = currNode;
            currNode = prevNode.getNext();
        }
        if (currNode.getValue() == v)
            return false; // v already exists
        else {
            newNode.setNext(currNode);
            prevNode.setNext(newNode);
            return true;
        } // if
    } // isolated
} // insert
```

Figure 8.1 : Naive linked list Insert with multi-place isolated statement.

will be thrown as a result of a failed place cast.) While single-place atomicity semantics are sound, they can be restrictive in many cases because they cannot be used to enforce atomicity across operations on data that reside in multiple places.

8.3 Multi-place Isolation

In this section, we describe an approach to support *multi-place isolation.* The main extensions to the X10 model can be summarized as follows.

Multi-place isolated statements: We introduce two new statements in addition to `isolated S`. The `isolated(`⟨*place-list*⟩`)` statement allows the scope of an isolated statement to span multiple places specified in *place-list.* Thus, the single-

place `atomic S` statement from X10, can be rewritten as `isolated (here) S`. The `isolated(*)` statement expands the scope of an isolated statement to all places. The Locality Rule is now generalized to requiring that all data accessed within a multi-place isolated statement be local to one of the places in the statement's scope. The motivation for this extension is that it enables programs to enforce isolation across multiple places.

Non-isolated remote accesses: We permit place remote accesses outside of isolated blocks. In the X10 model, such accesses had to be accomplished by creating new activities that run on the remote place. The motivation for this extension is that it does not require the programmer to ensure that all accesses outside an isolated block are local.

We use the Sorted Linked List example from [HLMS03] to illustrate multi-place isolation. Figure 8.1 shows a naive approach to implementing the `insert()` operation using the `isolated (*)` multi-place isolated statement. The advantage of this approach is that it is easy to verify its correctness; the disadvantage is that it can lead to a lot of unnecessary contention since the `while` loop for looking up the value is included in the scope of the `isolated` statement. Figure 8.2 shows an optimized approach using the ideas from the lazy concurrent list-based set algorithm in [HHL+05], but it uses a two-place* isolated statement to perform the list modification instead of relying on lower-level operations such as `CAS` (and the accompanying complexity of algorithms that use such operations).

We believe that multi-place isolation will be important for scalability on future many-core architectures, while still delivering the productivity benefits of using an

*This statement degenerates to a single-place isolated statement when both operands refer to the same place.

```
public boolean insert(int v) {
    INode newNode = new INode();
    newNode.setValue(v);
    while (true) {
        INode prevNode = this.first;
        INode currNode = prevNode.getNext();
        while (currNode.getValue() < v) {
            prevNode = currNode;
            currNode = prevNode.getNext();
        }
        if (currNode.getValue() == v)
            return false; // v already exists
        else
            isolated (prevNode.place, currNode.place) {
                if (validate(prevNode, currNode)) {
                    newNode.setNext(currNode);
                    prevNode.setNext(newNode);
                    return true;
                } // if
            } // isolated
     } // while
} // insert
```

Figure 8.2 : Optimized linked list Insert with multi-place isolated statement.

`isolated` construct in lieu of explicit lock operations. Both transactions and locks can be used as implementation techniques for multi-place isolated statements. In the next section, we introduce a two-level locking scheme for multi-place isolation, but we believe that similar ideas can also be used to improve the scalability of transactional memory implementations for multiple places.

```
globalLevelLock.writeLock().lock();
...;
globalLevelLock.writeLock().unlock();
```

Figure 8.3 : Implementation of isolated(*).

8.4 Two-level Lock-based Place Isolation

We use a two-level locking algorithm in our experiments to implement multi-place isolation semantics. The two-level lock is implemented as a global read-write lock and a set of place-level locks. Any `atomic(<place list>)` transaction that does not need to lock all places acquires the read lock of the global lock before acquiring the place locks it needs. The place locks are acquired following a global order to ensure deadlock freedom during execution. Any `atomic(*)` transaction that needs to acquire all the places must first acquire write access on the global lock. By obtaining an exclusive right to all places, `atomic(*)` transactions achieve the mutual exclusion with any other atomic(⟨place list⟩) or `atomic(*)`. Figures 8.3 and 8.4 outline the major components of our two-level locking algorithm.

Our two-level locking scheme shares some similarities with lock based software transactional memory systems [DSS06, SATH+06]. Lock-based STMs use locks as the underlying synchronization mechanism. In lock-based STM systems, mutual exclusion is guaranteed as a combined effort of the entire TM system which includes bookkeeping, validation, rollback, etc. The granularity of a lock-based STM can be a word, cache line, page, or an object. Currently our two-level locking scheme uses a place granularity; we will consider finer granularity in future work. Another characteristic of our two-level locking scheme is that the multi-place atomicity requires a

```
globalLevelLock.readLock().lock();
ForEach(Place_i in (sorted) places to lock)
{
    placeLevelLock[i].lock();
}
...;
ForEach(Place_i in (sorted) places to lock)
{
    placeLevelLock[i].unlock();
}
globalLevelLock.readLock().unlock();
```

Figure 8.4 : Implementation of isolated(⟨place list⟩).

pre-selected set of places. This set is either * for all places or a list of places that is explicitly specified. A dynamic-sized list of places is currently not supported in our two-level locking scheme.

8.5 Experimental Results

We chose to evaluate a Sorted Linked List implementation as the example data structure to evaluate the multi-place locking scheme. It is representative of linked lists used in many applications, and it has also been studied in quite some depth in many software transactional memory systems *e.g.,* [HLMS03]. We implemented the two-level locking scheme in Java, given that the current implementations of X10 are also based on Java.

We used a simple place assignment scheme for linked list nodes based on a uniform sub-range partitioning of the node values. Since the node values are generated by a random number generator, the uniform sub-range partitioning typically leads to a balanced partitioning of nodes among places. The nodes in the list are sorted in

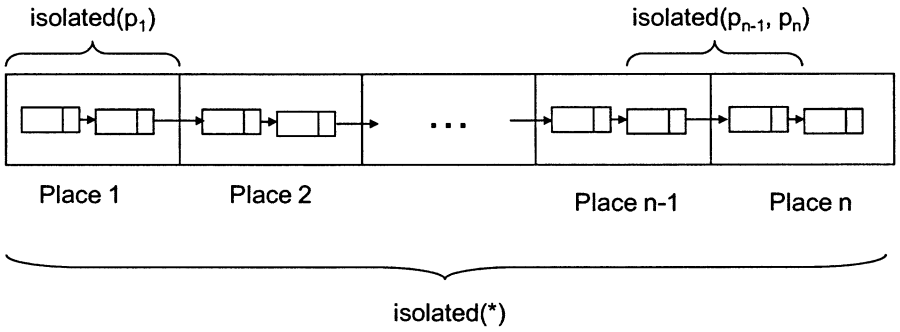


Figure 8.5 : Multi-place based sorted LinkedList implementation.

ascending order based on their values. There are four list operations: insert, remove, lookup, and sum. `insert(v)` inserts a new node with value `v` into the list if no node in the list has value `v`. `remove(v)` removes the node with value `v` if such a node exists in the list. `lookup(v)` checks if the list contains a node with value `v`. `sum()` returns the sum of all the values in the list[†]. `lookup()` is wait-free and does not use any per-place isolation. Insert and remove use `isolated(p)` or `isolated(p1,p2)` since depending on the location where the updates occur, these functions may operate on nodes in one or two places. Figure 8.5 shows how the list is mapped to places and in what situations `isolated(p)` or `isolated(p1, p2)` are used. These operations are implemented using the ideas from the lazy concurrent list-based set algorithm in [HHL+05], but they use multi-place isolated operations instead of lower-level synchronization primitives such as the optimized insert operation discussed earlier in Figure 8.2.

We evaluated the place-based list implementation on the following platforms, all

[†]In our experiments, `isolated(*)` is used to guarantee the isolation of the sum function though this is not strictly necessary.

with the Java `-server` option enabled:

- Gamma: Niagara 2, 8 cores, 8 hardware threads per core at 1.6GHz.
- Swym: SunFire 6800 server with 16 UltraSPARC III processors running at 1.2 GHz.
- Sugar: dual 2.83GHz Intel Xeon quad core processors (8 cores in total).

Figure 8.6, 8.7, and 8.8 summarize the performance results obtained on Gamma, Swym, and Sugar respectively. We measured the list throughput in 20 seconds for different numbers of threads, different numbers of places and different operation mixes on each system. The number of threads used on each platform was varied in powers of 2 from 1 to the number of hardware threads supported on that platform. These are 64, 16 and 8 for Gamma, Swym, and Sugar respectively. We varied the number of places in powers of 2 from 1 place to 4 times the number of hardware threads. We used three different operation mixes. The first is 5% insert, 5% remove, 89% lookup and 1% sum. The second is 25% insert, 25% remove, 49% lookup and 1% sum. The third is 45% insert, 45% remove, 9% lookup and 1% sum.

We can make several observations about the results of the experiments. First, the throughput scales up when increasing the number of places in most cases. This demonstrates the scalability advantage from using per-place isolation; however, the throughput improvement shows diminishing returns when the number of places exceeds the number of threads. The single place case in all these results is essentially equivalent to coarse grain locking for the entire list. The performance gap between 1 place (coarse grain locking) and P places for a P-way system is often largest for the P-thread case and for operation mixes that include more update (insert/remove) operations. These gaps were measured at up to 4.9× on Gamma, up to 2.6× on Swym,

and up to 2.1× on Sugar. In many cases, adding more threads led to a decrease in throughput if the number of places was smaller than P on a P-way system (due to coarseness in the locks), and if the hardware (*e.g.*, Sugar) was less scalable in its ability to support independent locking operations (most likely due to bus contention). However, in all cases, multi-place isolation outperformed single-place isolation.

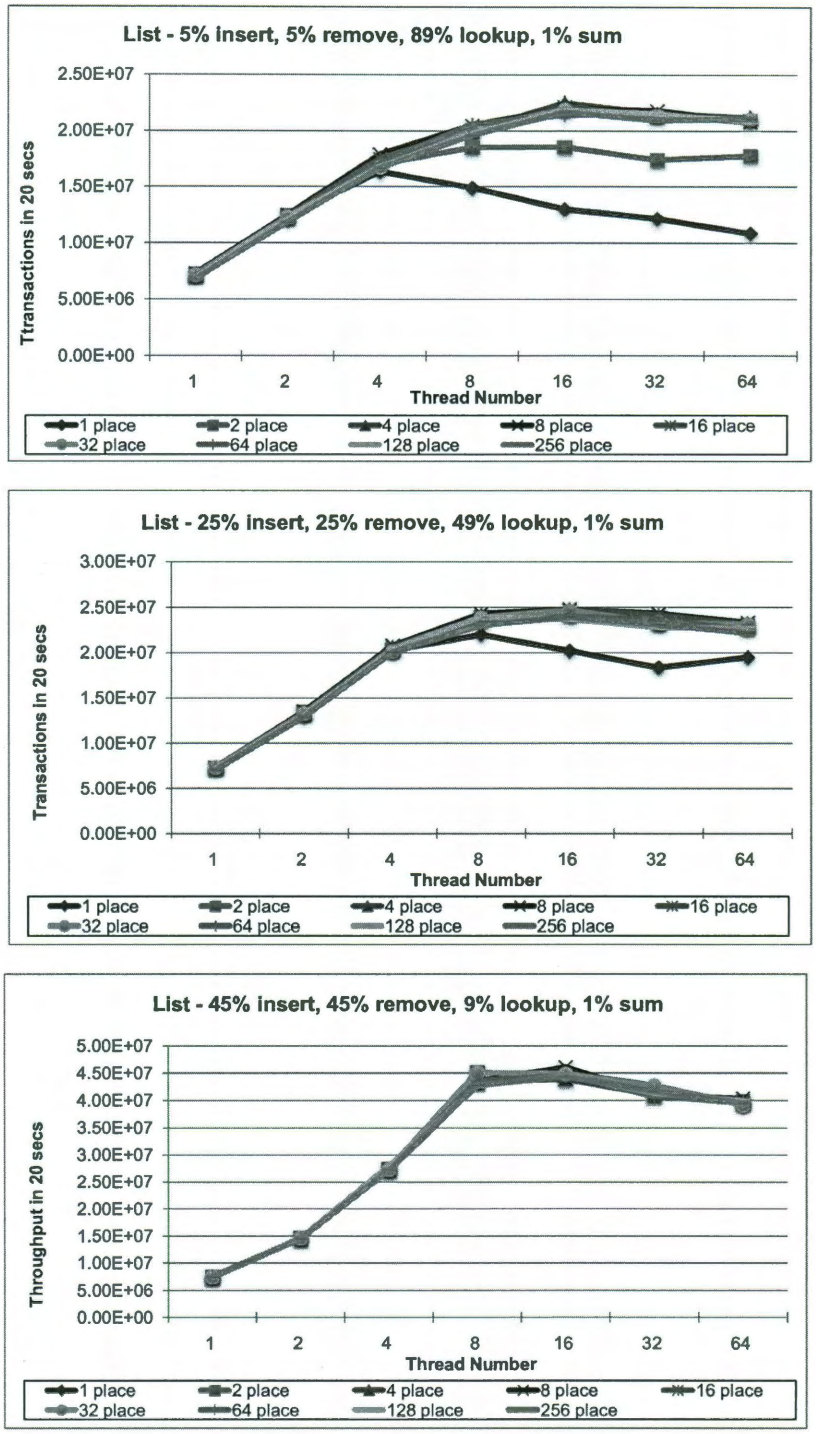


Figure 8.6 : Throughput on Gamma - a 64-way Niagara 2 system.

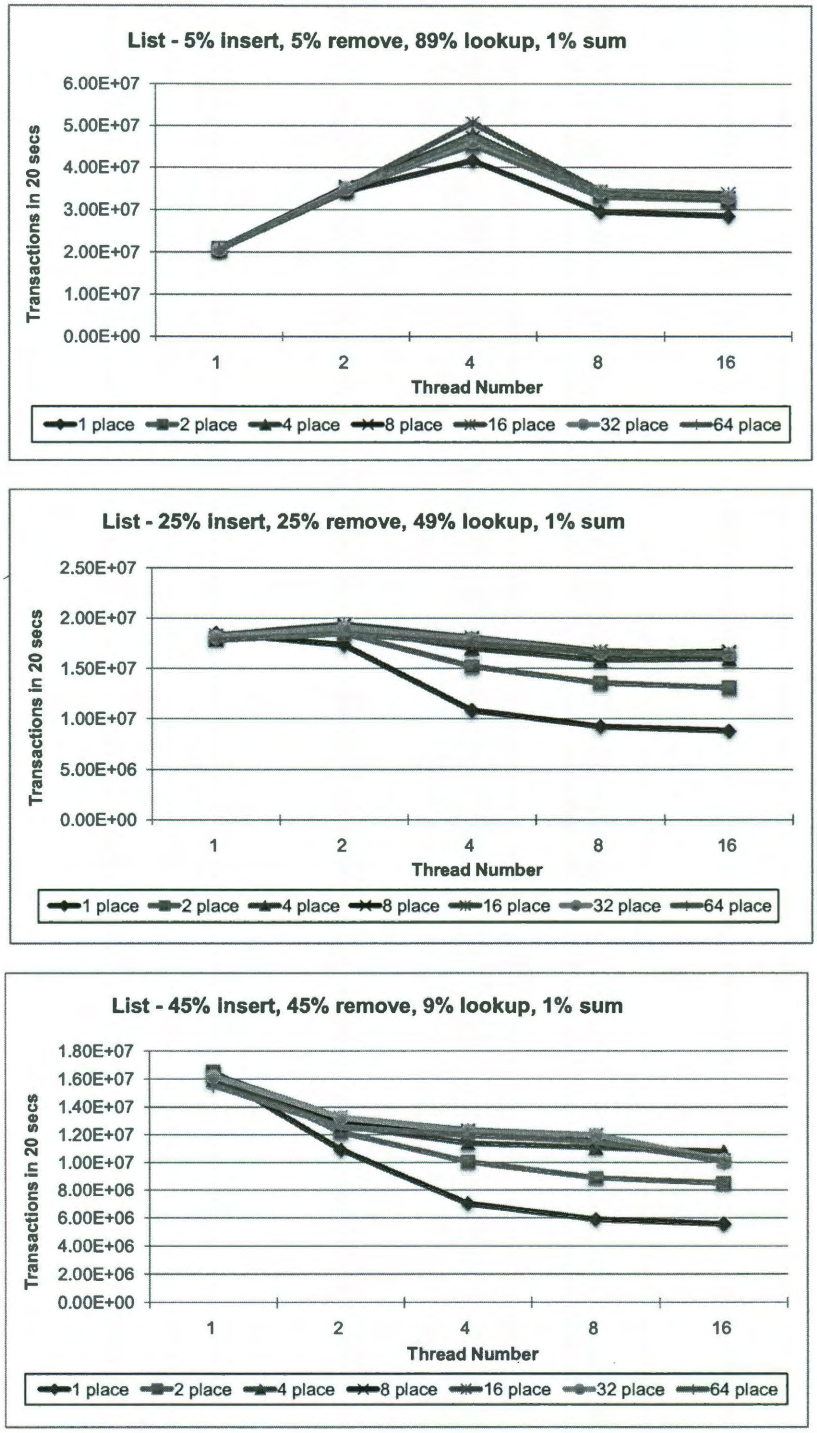
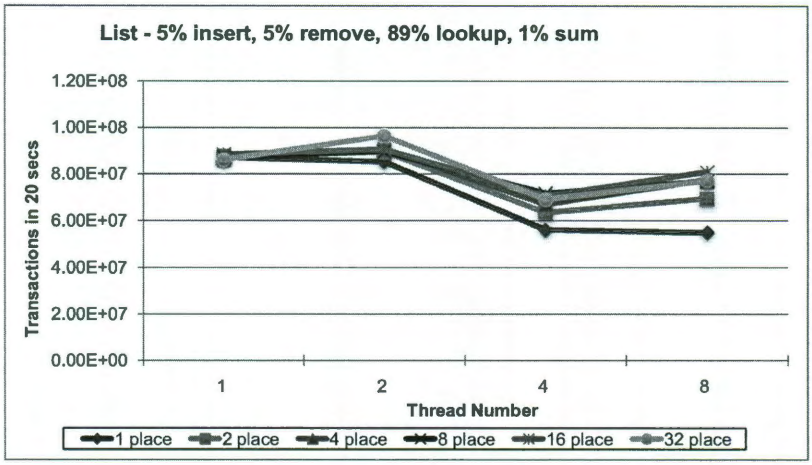
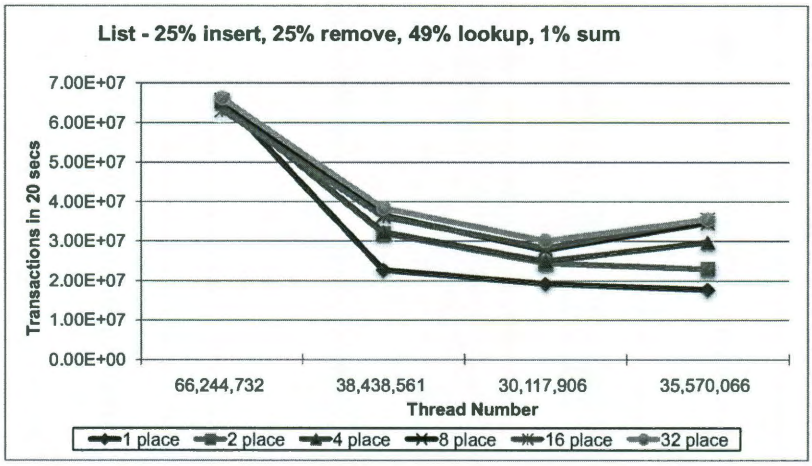


Figure 8.7 : Throughput on Swym - a 16-way SunFire 6800 system.





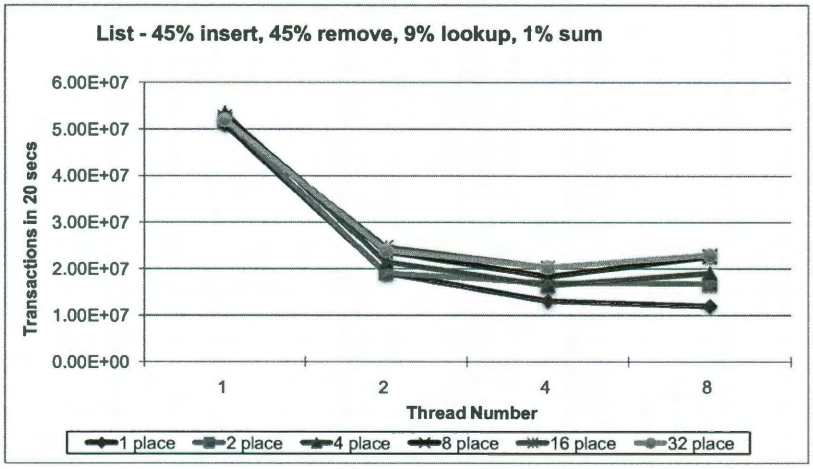


Figure 8.8 : Throughput on Sugar - an 8-way Intel Xeon system.

Chapter 9

Conclusions

First, we have presented a novel profile-driven strategy for runtime tuning of software transactional memory validation heuristics. We have described the design and implementation of such a strategy, and evaluated its effectiveness on a set of transactional memory benchmarks in scenarios with different contention levels, numbers of competing threads and read/write ratios.

We have shown that our strategy is competitive with the state-of-the-art validation techniques and that it performs on par with the best heuristic for any given constant contention level. We have also shown that in a scenario with dynamically changing contention levels, our strategy consistently outperforms the state-of-the-art techniques by up to 30% in the cases we have tested.

Second, we have presented an in-depth analysis of the commit phase in timestamp-based software transactional Memory implementations. We show that forcing the increment of the global shared counter in the commit phase may force unnecessary extra validation in some cases. We have also shown that performing the final validation for write transactions may be unnecessary in some cases.

We have presented several variants of the commit sequence that either avoid forced updates to the shared global counter (and thereby reduce contention on it), or avoid performing validation in cases when it is safe to do so, or both. We have shown that all the proposed variants of the commit phase preserve an important property of the STM systems: a transaction must never observe an inconsistent state of the shared

memory. We have also shown that the implementation of the global commit counter validation strategy in RSTM 2 has an invalid commit phase sequence that potentially allows transactions to observe an inconsistent state of the shared memory, and have proposed a different commit phase sequence that preserves consistency.

We have evaluated the proposed commit sequence variants on a set of transactional memory benchmarks, and shown variations that result in up to a 33% difference in overall system throughput.

Third, we have identified that common programmer practices in optimizing transactional applications by bypassing TM system calls and accessing the shared memory directly, in addition to breaking consistency and isolation (which have to be handled by the programmer manually), also break another desirable property of transactions: composability.

We have proposed two extensions to the TM system interface: $TxFastRead$ and $TxFlush$. These extensions enable the programmer to access shared memory in a more controlled manner, without breaking transaction composability. We have presented two techniques for implementing these system extensions: a lookup scheme and a partial commit scheme. We implemented these techniques on top of the TL II software transactional memory implementation and demonstrated on a set of benchmarks that we can obtain performance that is competitive with non-composable hand optimized code, while preserving composability. Compared to the existing practices, our system extensions require similar programmer effort (the programmer still needs to manually ensure isolation and consistency), provide similar performance, and yet preserve composability.

Fourth, we discussed several possible time base designs with a focus on the distributed time base design for STM systems. We focus on designs for multi-core

processors. Our experiments show that the single counter time base is a very competitive design; however, the distributed design is also a good alternative for some applications.

It is worth noting that there exist many different time base designs for software transactional memory systems. Theoretically each of them has its performance tradeoffs and could be a good fit for different application scenarios. Multi-core processors provide fast inter-core communication via on-die cache and network. From the results of this chapter, multi-core processors make centralized time base management very attractive. But even though the distributed time base design does not win broadly over a centralized single counter design, it still shows very competitive performance and gains small wins for several benchmarks. This makes the distributed time base an attractive alternative design choice to the centralized approach. For a system that contains multiple processors communicating via slower channels, the scalability bottleneck on the centralized time base will manifest itself more easily and make the distributed time base more attractive.

Fifth, we proposed a multi-place isolation approach to address the scalability challenges of enforcing atomicity when operating on distributed data. Our solution extends the X10 model with *multi-place isolated statements* of the form `isolated(`⟨*place-list*⟩`)` and `isolated(*)`, and generalizes X10's locality rule to require that all data accessed within a multi-place isolated statement be local to any one of the places in the statement's scope. The *two-level lock-based* implementation of multi-place isolation presented in this thesis is guaranteed to be deadlock-free and shows scalability improvements when increasing the number of places. For a simple Sorted Linked List example, we have observed the following improvements in throughput relative to a single place locking scheme on three different hardware platforms —

up to 2.1× on a dual quad-core Intel Xeon SMP with 8 cores, up to 4.9× on a Sun Niagara 2 SMP with 8 cores and 64 hardware threads, and up to 2.6× on a SunFire 6800 server with 16 UltraSPARC III processors.

In summary, this dissertation focuses on developing techniques to improve software TM performance in my dissertation. We show that the performance of STM can be significantly improved by intelligently designing validation and commit protocols, by designing the time base, and by incorporating application-specific knowledge based on our techniques. In particular, we (I) presented a time-based STM system based on a runtime tuning strategy that is able to deliver performance equal to or better than prior strategies, (II) designed several novel commit phase designs to improve the performance of existing commit protocols, (III) proposed a new STM programming interface extension that enables transaction optimizations using fast shared memory reads while maintaining transaction composability, (IV) presented a distributed time base design that outperforms existing time base designs for certain types of STM applications, and (V) proposed a novel programming construct to support multi-place isolation. We expect these techniques to help improve STM systems' performance and make STM be accepted by more programmers.

Bibliography

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[ACH+08] Eric Allan, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, March 2008.

[AR05] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005. In conjunction with OOPSLA'05.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[BLM05] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *An-*

nual Workshop on Duplicating, Deconstructing, and Debunking. Madison, Wisconsin, USA, Jun 2005.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[CH04] Christopher Cole and Maurice Herlihy. Snapshots and software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, Newfoundland, Canada, 2004.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, Seattle, Washington, USA, Sep 2008.

[Cra10] The Chapel language specification version 0.795. Cray Inc., Apr 2010.

[CSSB08] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on*

Principles and practice of parallel programming, pages 11–22, New York, NY, USA, 2008. ACM Press.

[CTTC06] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[DS06] David Dice and Nir Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.* Ottawa, Canada, Jun 2006.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006.* Springer, Stockholm, Sweden, Sep 2006.

[FM99] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *ICML '99: Proceedings of the Sixteenth International Conference on Machine Learning*, pages 124–133, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Fra03] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nine-*

teenth International Symposium on Distributed Computing, pages 303–323. LNCS, Springer, Sep 2005.

[Her90] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *PPoPP '90: Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, WA, USA, Mar 1990. ACM Press.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[HHL+05] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, Pisa, Italy, Dec 2005.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 26–37, Ottawa, Canada, Jun 2006.

[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[KCJ+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, New Yord, NY, USA, Mar 2006.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lee61] C. Y. Lee. An algorithm for path connection and its application. In *IRE Trans. Electronic Computer, EC-10*, 1961.

[Lie04] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[LM04] Yossi Lev and Mark Moir. Fast read sharing mechanism for software transactional memory (poster). In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing.* Las Vegas, Nevada, USA, Jul 2004.

[LMN07] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Portland, Oregon, USA, 2007.

[LR06] James R. Larus and Ravi Rajwar. *Transactional Memory.* Morgan & Claypool, 2006.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Austin, Texas, USA, Feb 2006.

[MH05] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and preliminary architecture sketches. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, USA, Oct 2005.

[MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 198–208. ACM Press, Manhattan, New York City, NY, USA, Mar 2006.

[Mos05] J. Eliot B. Moss. Open nested transactions: Semantics and support (poster). In *4th Workshop on Memory Performance Issues*, Austin, Texas, USA, Feb 2005.

[MSH+06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT 06': Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, Jun 2006.

[MSS04] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Houston, TX, Oct 2004.

[MSS05] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May 2005.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, New York, NY, USA, 2006. ACM Press.

[NMAT+07] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM Press.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM QUEUE Magazine*, Sep 2005.

[RFF06] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Stockholm, Sweden, Sep 2006.

[RFF07] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, San Diego, CA, USA, Jun 2007.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling

highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305. Austin, TX, USA, Dec 2001.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17. San Jose, CA, USA, Oct 2002.

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Madison, Wisconsin USA, Jun 2005.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[SJMvP07] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 161–172, New York, NY, USA, 2007. ACM.

[SMD+05] Arrvindh Shriram, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N.

Scherer III, and Michael F. Spear. Hardware acceleration of software transactional memory. Technical Report TR 887, Computer Science Department, University of Rochester, Dec 2005. Revised, March 2006; condensed version submitted for publication.

[SMSS06] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC06: 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, 2006.

[SMv08] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 275–284, Munich, Germany, Jun 2008.

[SPSS08] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer III. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomp uting*, pages 277–288, New York, NY, USA, 2008. ACM Press.

[SS05] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[SSHT93] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib.*

Technol., 1(4):58–71, 1993.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[TRA96] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 71–88, London, UK, 1996. Springer-Verlag.

[Vij08] Vijay Saraswat and Nathaniel Nystrom. Report on the Experimental Language X10 Version 1.7. http://www.npac.syr.edu/projects/cpsedu/summer98summary/ examples/hpf/hpf.html, September 2008.

[YNW+08] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 265–274, New York, NY, USA, 2008. ACM Press.

[ZB06] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.* Ottawa, Canada, Jun 2006.

[ZBS08a] Rui Zhang, Zoran Budimlić, and William N. Scherer III. Commit phase

variations in timestamp-based STM. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 326–335, Munich, Germany, 2008.

[ZBS08b] Rui Zhang, Zoran Budimlić, and William N. Scherer III. Runtime tuning of stm validation techniques. In *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, Boston, Massachusetts, USA, 2008.

[ZBS10] Rui Zhang, Zoran Budimlić, and William N. Scherer III. Composability for application-specific transactional optimizations. In *ACM SIGPLAN Workshop on Transactional Computing*, Paris, France, 2010.

[ZSZ+08] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *21th International Workshop on Languages and Compilers for Parallel Computing*. Edmonton, Canada, 2008.